NASA/CR-1999-208971

Detecting Mode Confusion Through Formal Modeling and Analysis

Steven P. Miller and James N. Potts
Rockwell Collins, Inc., Cedar Rapids, Iowa

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Contract NAS1-19704

January 1999

Abstract

Aircraft safety has improved steadily over the last few decades. While much of this improvement can be attributed to the introduction of advanced automation in the cockpit, the growing complexity of these systems also increases the potential for the pilots to become confused about what the automation is doing. This phenomenon, often referred to as *mode confusion*, has been involved in several accidents involving modern aircraft. This report describes an effort by Rockwell Collins and NASA Langley to identify potential sources of mode confusion through two complementary strategies. The first is to create a clear, executable model of the automation, connect it to a simulation of the flight deck, and use this combination to review of the behavior of the automation and the man-machine interface with the designers, pilots, and experts in human factors. The second strategy is to conduct mathematical analyses of the model by translating it into a formal specification suitable for analysis with automated tools. The approach is illustrated by applying it to a hypothetical, but still realistic, example of the mode logic of a Flight Guidance System.

Acknowledgements

The authors thank Rick Butler and Victor Carreno of the NASA Langley Research Center for their help in organizing the PVS specification of the Flight Guidance System, formally characterizing some of the potential sources of mode confusion, and completing the proofs. The authors also thank the many engineers at Collins that have helped them in gaining a basic understanding of Flight Guidance Systems.

Contents

Figures

Chapter 1

Introduction

Aircraft safety has improved steadily over the last few decades [23]. While much of this improvement can be attributed to the introduction of advanced automation in the cockpit [2], the growing complexity of these systems also increases the potential for the pilots to become confused about what the automation is doing. Of particular concern is the proliferation of modes in these systems, where modes are defined as mutually exclusive sets of system behavior [13]. For this reason, the phenomenon is often referred to as "mode confusion".

There is a growing body of evidence that mode confusion is a legitimate concern in complex automated systems in which humans play a significant role. Several accidents and incidents involving mode confusion in modern aircraft are listed in [9]. A study conducted by the Massachusetts Institute of Technology found 184 incidents attributed to mode awareness problems in NASA's Aviation Safety Reporting System (ASRS) [9]. In [8], the author describes the concerns of pilots and researchers with the human computer interface in modern "glass cockpits". The FAA recently hosted a workshop on Autoflight Mode Awareness that identified "autoflight mode confusion as a significant safety concern" [1]. In [2], Charles Billings writes (pg. 183):

> *Most of our accidents can be traced to the human operators of the systems, and increasing numbers can be traced to the interactions of humans with automated systems.*

This paper describes an effort by Rockwell Collins and NASA Langley to identify potential sources of mode confusion through modeling and analysis of the automation. This approach makes use of two complementary strategies. The first is to create a clear, executable model of the automation, connect it to a simulation of the flight deck, and use this combination to review of the behavior of the automation and the man-machine interface with engineers, the pilots, and experts in human factors. This has several benefits. First, it forces the designers to commit to a clear, conceptual model of the automation. Second, it facilitates discussion between the system designers, experts in human factors, and the flight crew. Third, can be used in training to convey an accurate mental model of the automation to the flight crew.

The second strategy is to conduct mathematical analyses of the model. This is accomplished by translating the model into a formal specification suitable for analysis with automated tools. This model can be used to show that safety properties hold for all states reachable by the model. It is also possible to characterize some sources of mode confusion as mathematical statements about the model and use these tools to automatically find these potential sources.

The approach is illustrated by applying it to an example specification of the mode logic of a Flight Guidance System created by Collins to investigate different methods of modeling requirements. While this example is hypothetical and does not describe an actual aircraft in service, it is still complex enough to serve as a realistic example [15].

The rest of this report is organized as follows. Chapter 2 provides background information, including a brief description of related work and an overview of Flight Control Systems. Chapter 3 discusses the motivation behind this project and provides an overview of the approach. Chapter 4 discusses the rationale for the structure of the mode logic, illustrates the behavior of the model, and describes potential sources of mode confusion found in the example. Chapter 5 describes the formal model of the mode logic created in PVS, while Chapter 6 discusses some of the properties proven about this model. Finally, Chapter 7 summarizes conclusions and identifies possible future directions.

Chapter 2 Background

This chapter describes related work, provides a brief overview of a Flight Guidance System (FGS), and discusses the history of the FGS used as the example in this report.

2.1 Related Work

In [12], Nancy Leveson describes how operators create *mental models* of a system in order to understand and predict its responses to their inputs and the environment. Mode confusion is a discrepancy between the operators perceived and actual state of the automation, either due to a faulty mental model (i.e., a mental model that is not an accurate abstraction of the system) or due to the operator losing track of the state of the automation. In either case, the outcome can be one or more inappropriate commands to the system.

Leveson also points out how different individuals may form different mental models of the automation, based on their experiences and needs. For example, the designer may form a mental model based on mathematical models appropriate for situations where important decisions need not be made quickly, while the operator may form a mental model that emphasizes making diagnoses and responses quickly.

In [13], Leveson, et.al., discuss an approach to detecting error-prone automation features through modeling and analysis of the software. They also identify six categories of design that have historically been sources of mode confusion:

1. Interface interpretation errors
2. Inconsistent behavior
3. Indirect mode changes
4. Operator authority limits
5. Unintended side effects
6. Lack of appropriate feedback

The importance of an accurate mental model of avionics systems by the flight crew has been recognized by several researchers. Charles Billings argues for human-centered automation designed to work cooperatively with pilots and air traffic controllers in the pursuit of stated objectives [2]. While acknowledging that automation has done much to make aircraft safer, he points out that new classes of problems have emerged due to failures in the human-machine interface. He states (pg. 4):

In particular, we have seen the appearance of incidents and accidents that indicate failures to understand automation behavior. We have seen errors in choice of operating modes, lack of mode awareness, and inability to determine what the automation was doing.

Billings also identifies some of the sources of mode confusion. These include;

1. Complexity (particularly in the number of modes and interactions between modes)
2. Brittleness (inability to respond correctly at the margins of the operating range)
3. Opacity (inadequate display of what the automation is doing)
4. Literalism (inability to respond correctly to unanticipated situations)
5. Training (lack of training of automation behavior)

This last point emphasizes the need to convey to the pilot an accurate mental model of the automation. On (pg. 146), he states;

An adequate internal model of an automated system is vital to a pilot's ability to predict how that system will function under novel circumstances.

Other researchers have concurred on this point. In [20], Sarter and Woods write:

What is needed is a better understanding of how the machine operates, not just how to operate the machine.

Of particular concern seems to be the modes that control the vertical behavior of the aircraft. An MIT study described in [9] examined 184 incidents attributable to mode awareness problems found in NASA's Aviation Safety Reporting System (ASRS). They found that 74% of the errors involved confusion in vertical navigation, while only 26% involved horizontal navigation. The researchers attribute this to lack of appropriate feedback on vertical navigation. Another major factor is the greater complexity of vertical navigation, which involves different combination of elevator and thrust controls.

Tony Lambregts, the FAA National Resource Specialist for Automated Controls, agrees that the interaction of elevator and thrust controls for vertical navigation is an important source of mode confusion, but argues that the problem is not so much lack of consideration of human factors as adherence to outdated designs [11]. Flight Control Systems have evolved from the earliest designs of the Wright brothers through the gradual introduction of new control functions, where the control laws are based on single input single output (SISO) designs. For example, control of the Autopilot and Autothrottle are not tightly integrated at the innermost level. Instead, this integration is achieved at higher levels in the Flight Control System. Lambregts argues that since the response of the aircraft to pitch and thrust commands are inherently coupled by the dynamics of flight, this results in a proliferation of modes in order to achieve the desired behavior. By integrating control for inherently coupled functions through the use of multiple input multiple output (MIMO) control laws, he claims the Flight Control System can be greatly simplified, with a substantial reduction in system modes and potential for mode confusion.

2.2 An Overview of Flight Guidance Systems

A *Flight_Guidance System* (FGS) is a component of the overall *Flight Control System* (FCS) (see Figure 1). The FGS compares the measured state of an aircraft (position, speed, and attitude) to the desired state and generates pitch and roll guidance commands to minimize the difference between the measured and desired state. When engaged, the *Autopilot* translates these commands into movement of the aircraft's control surfaces necessary to achieve the commanded changes about the lateral and vertical axes.

An FGS can be further broken down into the *mode logic* and the *flight control laws*. The mode logic accepts commands from the flight crew, the *Flight Management System* (FMS), and information about the current state of the aircraft to determine which system modes are *active*. The active modes in turn determine which flight control laws are used to generate the pitch and roll guidance commands. The active lateral and vertical modes are displayed (annunciated) to the flight crew on the *Flight Director*, a portion of the Electronic Flight Instrumentation System (EFIS). The magnitude and direction of the lateral (roll) and vertical (pitch) commands generated by the FGS are also displayed on the EFIS as *guidance cues*.

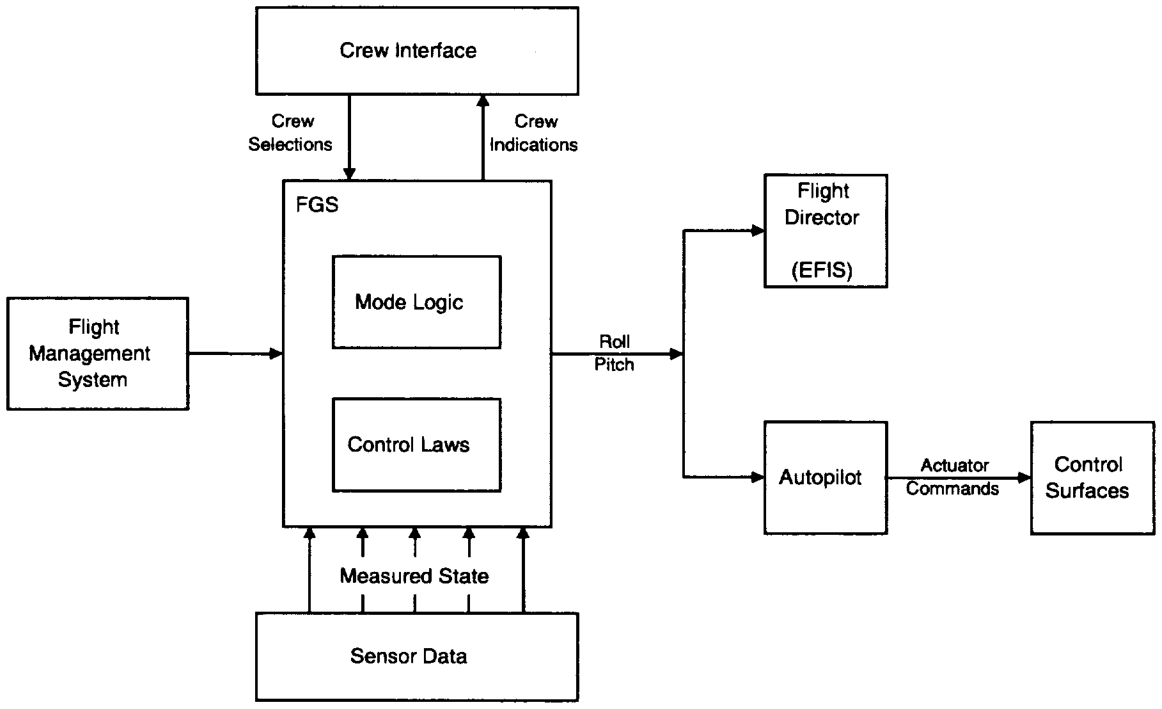


Figure 1 - Flight Control System

2.3 The Example Flight Guidance System

The approach described in this report was investigated using an example specification of the mode logic of a Flight Guidance System for a business jet/commuter class aircraft. This example was created by Collins to investigate different techniques for requirements modeling and analysis. The original specification was created using the CoRE methodology and has been placed in the public domain. It is fully described in [15]. Since that time, the same example has been translated into Z [5], SMV [17], SCR [16], T-VEC [3], and now ObjecTime [21] and PVS [18]. When determining how to model specific features in ObjecTime and PVS, the original CoRE specification [15] was used as the standard of "correct" behavior.

The original specification was an example derived through study of several Flight Guidance Systems. Moreover, to keep the size of the example tractable, it was simplified in several ways. First, it was restricted to specifying only the mode logic of the FGS. Second, several more complex features found in recent aircraft, such as the more complex vertical navigation modes, were omitted. Third, the example deals almost entirely with "normal" behavior and does not specify how the system should respond to internal errors such as failed sensors. Finally, the example does not specify the hardware interfaces (i.e., the CoRE input and output variables and the IN and OUT relations). Despite these limitations, the specification is sufficiently rich that it meets the goal of providing a realistic industrial example for the evaluation of other methods, including those described in this report. However, it is important to note that it does not describe an actual or planned Collins product.

Chapter 3
Overview of the Approach

3.1 Motivation

This project was motivated by several goals. These included creating a high level, dynamic model of the automation's behavior, performing automated analyses of this model, and supporting a product family approach to development. In some cases, these goals were in conflict and decisions had to be made regarding which were most important. These issues are discussed more fully in the following sections.

3.1.1 Visualization of the Automation

The main objective of this project was to provide a clear, comprehensible view of the underlying automation that could be executed. Our belief was that connecting this model to a mock-up of the Flight Control Panel (FCP) and the Electronic Flight Instrumentation System (EFIS) would provide a common focus that would promote discussion between pilots, experts in human factors, and the system designers. Our experiences to date have shown that, if anything, we underestimated the power of this technique. In every demonstration, the visualization has generated vigorous, positive debate between these groups.

A secondary objective was to force the development and commitment to a high level design of the automation. In projects developed without such a vision, design choices may be based on local concerns, such as fixing the immediate problem at hand or achieving a certain level of performance. This often results in unnecessary complexity that is confusing to both the users and developers of the system. Moreover, this complexity tends to grow as the system evolves over time. Having a clear, high level model of the automation encourages the developers (and customers) to make changes consistent with this model as time progresses. [1]

Finally, we were interested in whether such a dynamic, high level model of the automation would be of value during training of the flight crew. Would the availability of such a model enable the flight crew to obtain a better mental model of the automation? Would it allow them to internalize patterns so that they would be better able to predict the behavior of the system? For existing systems, could such models be used to explain confusing behavior more clearly? Or would the availability of such models simply add more complexity to pilots already overwhelmed with details?

[1] Of course, this doesn't exclude explicitly stating the design guidelines. In fact, they may be easier to state in the presence of a high level model.

3.1.2 Support for Automated Analysis

The value of automated analysis of properties such as consistency and certain forms of completeness has been clearly demonstrated in a number of different tools [6], [7], [16]. Other projects have shown how application specific properties, such as safety properties or desirable system behaviors, can be verified through the use of theorem provers [18], [14], and model checkers [17]. However, there are very few examples of using these techniques to check for properties related to human factors. For example, responding differently to a button press in different modes may not be an error, but it would be nice to know where such behavior is present in a model. The most interesting questions here are how such properties can be stated in a formal system and what sorts of properties relevant to human factors are amenable to automated analysis. Obtaining a least a partial answer to these questions was another important goal.

3.1.3 Support for Product Families

Finally, if complex systems are to be affordable, planning for change and reuse has to play a larger role in the future. Companies typically build variations of the same products over and over and are looking for strategies that support the systematic reuse of common artifacts.

One such approach is Product Family Engineering, also known as Domain Engineering. Product Family Engineering is a methodology that focuses on creating software and hardware assets that can be systematically reused in each new member of the product family [4], [22], [24]. Central to the Product Family approach is the development of a domain architecture. The domain architecture consists of those requirements, design, implementation, and verification artifacts that are common to all members of the family and the variations of these artifacts that are supported by the domain. New instances of the product family (applications) are created by selecting from the assets already available and supplementing and tailoring them as needed to create a specific product. New product specific artifacts are carefully factored back into the domain architecture to enrich the base of available assets.

Prior to this project, Collins had conducted a Commonality Analysis [24] of the FGS mode logic described in [15] and developed a tentative product family architecture for the FGS. Consequently, an important goal of this project was to build on that work and develop a formal model consistent with that architecture.

3.1.4 Logistics

Finally, experience with previous demonstrations had convinced us that we wanted a very stable, portable demonstration that we could pick up and take anywhere at a moment's notice. To ensure widespread availability of the work products, we wanted to avoid the use of proprietary tools or components that were not available to the general public. We also realized having sufficient screen area to display both the visualization of the automation and the mockup of the flight deck would be essential.

3.2 Approach

The approach taken in this project directly addressed the issues raised in the previous section and is illustrated in Figure 2. Executable model were used to drive visualizations of both the Flight Deck (i.e., the Flight Control Panel and the EFIS) and the internal state behavior of the automation. Formal models were derived from the executable models and used for formal analysis with a theorem prover. This approach is discussed in greater detail in the following sections.

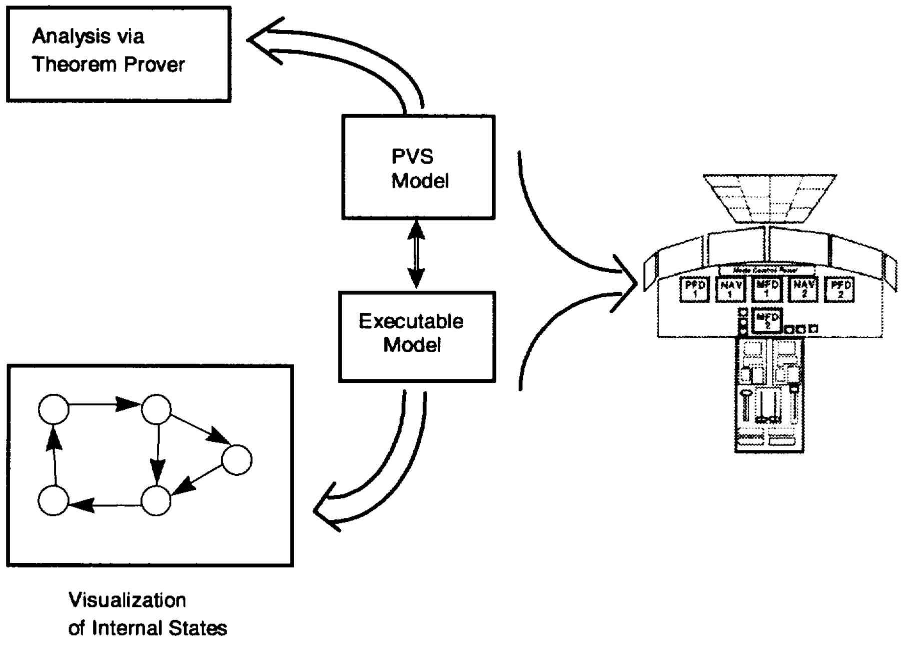


Figure 2 - Overall Strategy

3.2.1 Visualizations

ObjecTime [21] was chosen as the modeling environment since the ObjecTime models could be executed and easily connected to visualizations of the Flight Control Panel and EFIS. ObjecTime also has a Statecharts-like capability for visualizing the behavior of the models that we hoped could be used directly in demonstrations. Finally, we were confident that ObjecTime was compatible with the product family architecture previously developed for the FGS.

The visualizations of the Flight Control Panel and the Electronic Flight Instrumentation System were among the simplest part of the project to create. Ours were created using Borland Delphi [2].

The visualization of the Electronic Flight Instrumentation System (EFIS) is shown in Figure 3. The EFIS is the primary flight display in the cockpit. It displays essential information about the aircraft, such as airspeed, vertical speed, attitude information, the horizon line, and heading. Particularly relevant for this project are the flight mode annunciations at the top of the display (e.g., HDG and PTCH), the AP engaged annunciation located at the upper left of the sky/ground ball, and the guidance cues shown in the center of the sky/ground ball. The combination of the flight mode annunciation and the guidance cues are often referred to as the Flight Director. Only the mode annunciations and the guidance cues needed to be active components of the EFIS visualization for this project.

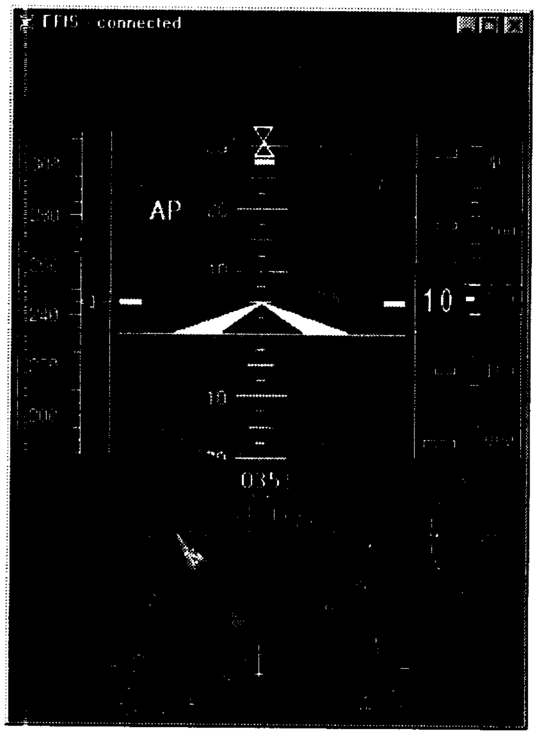


Figure 3 - EFIS

The visualization of the Flight Control Panel is shown in Figure 4. The Flight Control Panel is the primary user interface with the Flight Control System. It includes switches for turning the Flight Director on and off (FD), switches for selecting the different flight modes such as vertical speed (VS), lateral navigation (NAV), heading hold (HDG), altitude hold (ALT), and approach (APPR), the Vertical Speed/Pitch Wheel, and the autopilot disconnect bar. The FCP also supplies feedback to the crew, indicating selected modes by lighting lamps on either side of a selected mode's button.

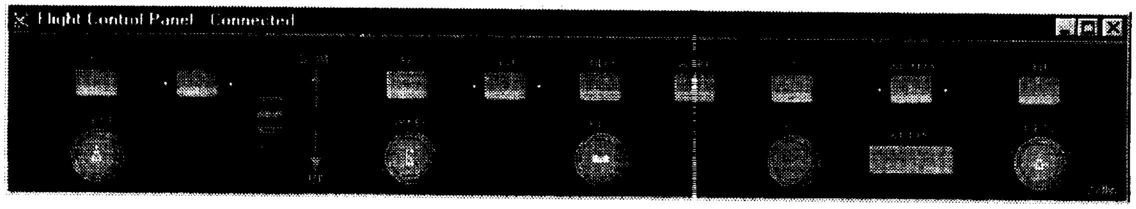


Figure 4 - Flight Control Panel

[2] Borland Delphi is a trademark of the Inprise Corporation.

As the project progressed, it became clear that the ObjecTime diagrams were too cluttered with design detail to use for communication with pilots and experts in human factors. This was easily remedied by creating our own visualization of the automation and using the ObjecTime models to drive it as well as the FCP and EFIS visualizations. An example of this visualization is shown in Figure 5. High level views of the mode machines for the Flight Director, Autopilot, and each of the lateral and vertical flight modes are shown. The current mode of each mode machine is indicated by turning the mode a bright green (shown here as white). For example, in Figure 5 the Flight Director is in state *Cues On*, the Autopilot is in state *Engaged*, the lateral navigation (NAV) mode is in state *Armed*, the vertical speed (VS) mode is in state *Active*, and all other lateral and vertical modes are in the state *Cleared*. Certain states are distinguished as *active* states. When in these states, the associated mode is also said to be active and generates the pitch and roll guidance commands used as inputs to the autopilot and the flight director. Stated differently, if the Autopilot is engaged, the active lateral mode generates roll guidance commands and the active vertical mode generates pitch guidance commands used to control the aircraft. Active states are indicated on the visualization by a heavy red box (shown here as a gray box) around the state or group of states. A few of the active states in Figure 5 include the *Active*

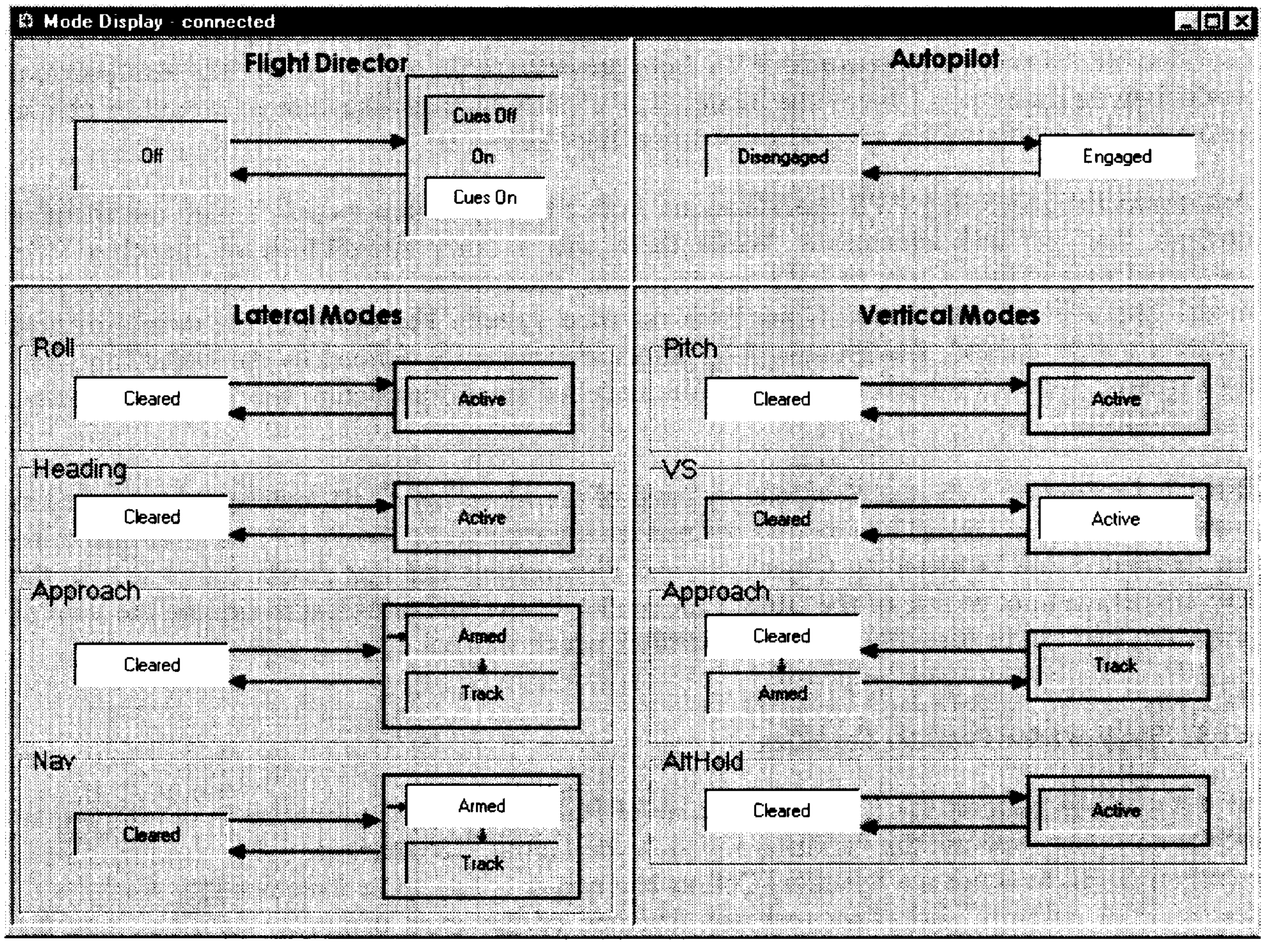


Figure 5 - Visualization of the FGS Modes

state of Roll mode, the *Armed* and *Track* states of lateral Approach mode, and the *Track* state of vertical Approach mode. As will be discussed later, only one lateral and one vertical mode can be active at any given time. This property is emphasized in the visualization by placing the active lateral states directly below each other. The active vertical states are laid out in a similar fashion. In Figure 5, the active lateral mode is NAV and the active vertical mode is VS.

Having the capability of designing our own visualization of the automation turned out to be a significant advantage. Since our focus was on conveying a very high level mental model, we ended up removing details that obscured that model, but that would be necessary to fully specify the system. Also, reviewers began suggesting changes to the model specifically for the purpose of conveying the mental model, e.g., changing the color of all transitions just executed and graying out transitions that were not possible. This made it clear that designing mental models of the automation is an open area for further work.

3.2.2 Support for Automated Analysis

PVS was chosen for analyzing our models for application specific properties because it provides a very powerful and general mechanism for stating and proving properties [18]. Also, we had considerable in-house expertise with PVS from previous projects [14]. There was also previous work done on translating ObjecTime models to PVS that gave us hope that we would be able to automatically generate PVS specifications from ObjecTime [10].

Automatically generating PVS specifications from the ObjecTime models turned out to more difficult than we had anticipated. While there was a fairly straightforward mapping from individual ObjecTime actors to PVS specifications, these were of limited value unless a PVS model was also created for the ObjecTime run-time system. For example, the details of how events are sent, queued, and processed by actors are implicitly defined by the ObjecTime run-time system. We decided that creating a PVS model of this infrastructure was beyond the scope of the project.

Instead, we created PVS specifications by hand of the mode logic and reviewed them to try to ensure the PVS and ObjecTime models defined the same behavior. While it is true that this broke the automated link between the ObjecTime and PVS models and that proofs completed for the PVS model may not be true of the ObjecTime model, it was still sufficient to explore the sorts of properties that can be formally stated and verified, which was our original goal.

3.2.3 Support for Product Families

As discussed in Section 3.1.3, the cornerstone of Product Family engineering is developing an architecture that supports the variation found in the family. To determine what variations were common in Flight Guidance Systems, Collins had earlier conducted a Commonality Analysis of several FGS systems [24]. This revealed that one of the most common variations was the configuration of operational modes. Both the complement of operational modes installed and the versions of individual modes varied from aircraft to aircraft. Thus, one aircraft might have lateral backcourse mode and the next aircraft might not. Two aircraft might both have lateral navigation modes, but have slightly different versions of that mode.

However, there were also many properties common to all aircraft. In the systems studied, every mode was either a lateral mode that controlled the aircraft about the roll axis or a vertical mode that controlled the aircraft about the pitch axis. Every mode could either be active or inactive. There could never be more than one lateral mode active at any time, and there could never be more than one vertical mode active at any time. If the Flight Director was turned on, one lateral mode and one vertical mode had to be active. There was always a default lateral mode and default vertical mode that became active when the Flight Director was turned on or when the active lateral or vertical mode was made inactive.

To exploit this commonality and support this variation, this project adopted an architecture for the Flight Guidance System that made it straightforward to produce different configurations of operational modes. This architecture is described in detail in Chapter 4.

Chapter 4

Structure of Mode Logic

The chapter provides an informal, intuitive presentation of the FGS mode logic. Since the architecture of the FGS also affects presentation of the mode logic, it first discusses the rational for this architecture. It then describes the behavior of the mode logic with a few examples. These are typical of scenarios that can explored with the design engineers, the flight crew, and human factor experts using the executable model and the visualizations. Finally, a few potential sources of mode confusion found in the example FGS specification [15] are shown to demonstrate the effectiveness of combining an executable model of the automation with the visualizations.

4.1 Rational for the FGS Architecture

As discussed in Chapter 3, support for a family of Flight Guidance Systems was one of the primary factors in selecting the FGS architecture shown in Figure 6. At the lowest level, each operational mode is treated as its own unit of functionality, with no knowledge of the properties, or even the existence, of the other modes within the FGS. Each operational mode exports an indication of whether it is active, whether it is armed, and if it is active, the guidance command (roll for lateral modes and pitch for vertical modes) that it is computing. The lateral modes are grouped into the Lateral Guidance component and the vertical modes are grouped into the Vertical Guidance component.

The Lateral Guidance component enforces all constraints between the lateral modes. For example, it ensures that one lateral mode is active when the Flight Director is on and ensures that no more than one lateral mode is ever active. Any aircraft specific constraints between the lateral modes are also enforced by Lateral Guidance. Lateral Guidance also exports the identify of the current active lateral mode, the identity of all armed lateral modes, and the lateral guidance command generated by the active lateral mode. Vertical Guidance serves a similar function for the vertical modes, while the Flight Director component maintains the status (on, off, cues displayed, cues hidden) of the Flight Director.

All of these are grouped within the Flight Guidance component. Flight Guidance exports the status of the Flight Director, the identity of the current active lateral and vertical modes, the identity of all armed lateral and vertical modes, and the lateral and vertical guidance commands. Flight Guidance also enforces constraints between the Flight Director and the Lateral and Vertical Guidance components. For example, it ensures that one lateral mode and vertical mode are active when the Flight Director is turned on. It also enforces any constraints between components that are aircraft specific. For example, a common constraint is that lateral Go Around (GA) mode is active if, and only if, vertical Go Around mode is active

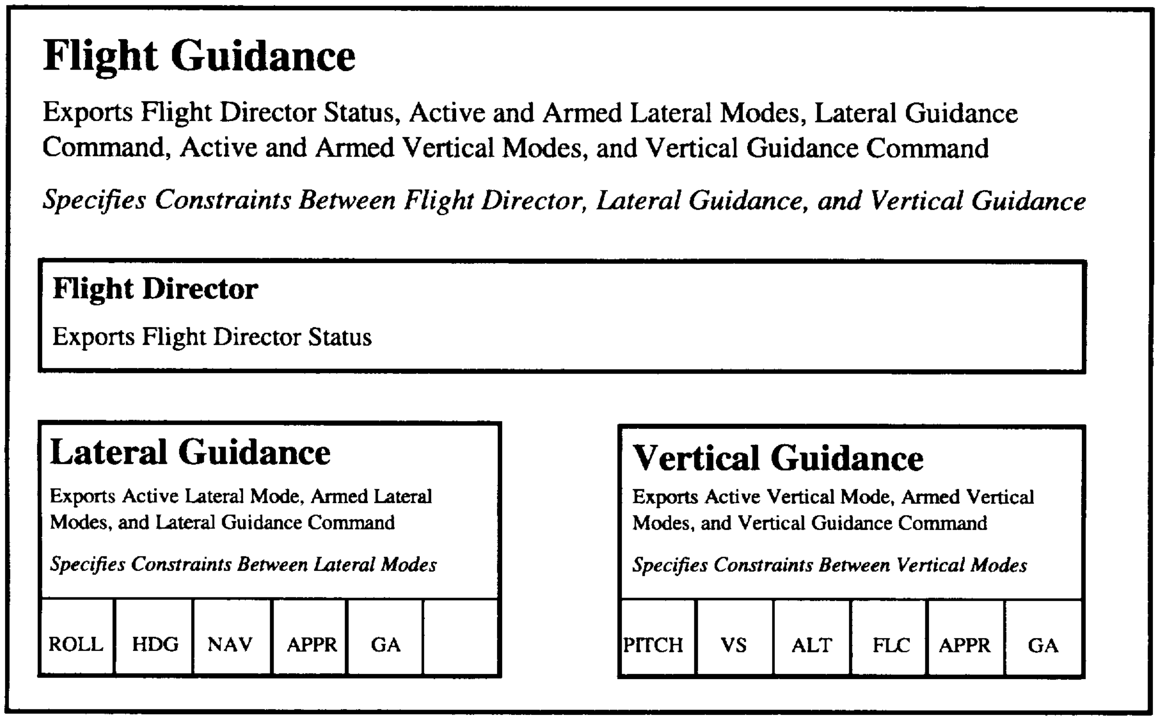

Flight Guidance
Exports Flight Director Status, Active and Armed Lateral Modes, Lateral Guidance Command, Active and Armed Vertical Modes, and Vertical Guidance Command
Specifies Constraints Between Flight Director, Lateral Guidance, and Vertical Guidance
Flight Director
Exports Flight Director Status
Lateral Guidance
Exports Active Lateral Mode, Armed Lateral Modes, and Lateral Guidance Command
Specifies Constraints Between Lateral Modes
ROLL
HDG
NAV
APPR
GA
Vertical Guidance
Exports Active Vertical Mode, Armed Vertical Modes, and Vertical Guidance Command
Specifies Constraints Between Vertical Modes
PITCH
VS
ALT
FLC
APPR
GA

Figure 6 - High Level FGS Architecture

The motivation behind this architecture is to minimize maintenance costs and to maximize reuse between different members of the product family. This is achieved by ensuring:

- the lateral and vertical modes all present similar interfaces to their parent
- a mode does not know about the properties or existence of other modes in the aircraft
- constraints between the siblings are localized in the appropriate parent.

In the following sections, the mode logic of a single member of this family, based on the example described in [15], is described in the context of this architecture.

4.2 Synchronization of the Mode Machines

Building upon the high-level architecture of the previous section, the FGS mode logic can most easily be visualized as a collection of tightly synchronized, concurrent mode machines, one for each component of the FGS (see Figure 7).

Most of the synchronization between the mode machines can be achieved by enforcing three simple properties:

- If the flight director is on, one and only one lateral mode is active.
- If the flight director is on, one and only one vertical mode is active.
- If the flight director is off, all lateral and vertical modes are cleared.

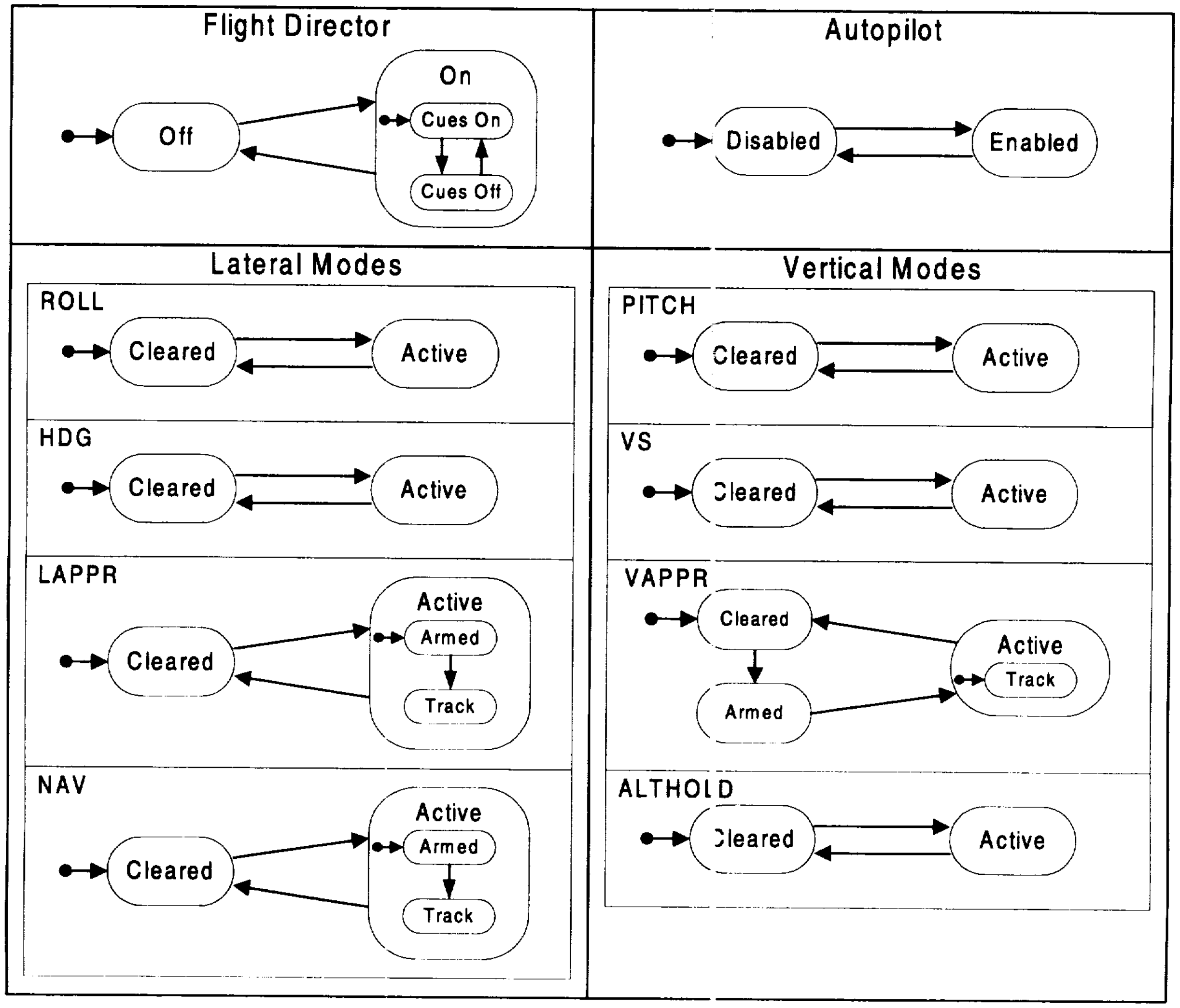


Figure 7 - FGS Mode Structure

These properties state that when the flight director is turned on, there must be one active lateral mode and one active vertical mode to produce roll and pitch commands. Obviously, there can be only one active lateral mode and one active vertical mode controlling the aircraft at any given time.

To satisfy the first two properties, two of the mode machines, lateral Roll and vertical Pitch, are designated as *default* modes. This designation means that if the Flight Director is on, and no other modes are active, these two machines will be active. For example, if HDG mode is active in the current state, and the HDG button is pressed on the Flight Control Panel, HDG will clear. Since one lateral mode must be active, the default mode (ROLL) will become active (See Figure 8).

Likewise, with ROLL active, if the HDG button is pressed again, HDG will be activated. Since only one lateral mode can be active, the previous active mode (ROLL) will be cleared (see Figure 9).

These synchronizations have all been between the Flight Director and the Lateral and Vertical modes, or within the Lateral and Vertical modes. More difficult are the synchronizations between the Lateral and Vertical modes. In the CoRE FGS specification, Lateral Approach (LAPPR) and Vertical Approach (VAPPR) are synchronized, so that when LAPPR enters Track, VAPPR becomes Armed (see Figure 10 through Figure 12).

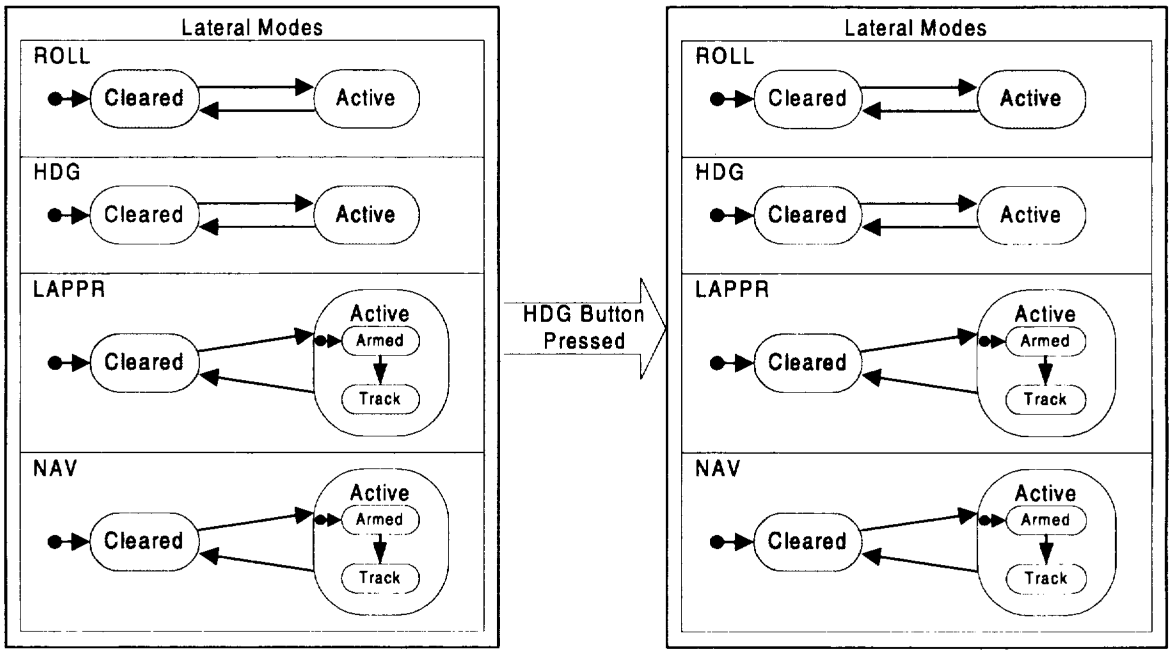


Figure 8 - Activation of Default Mode

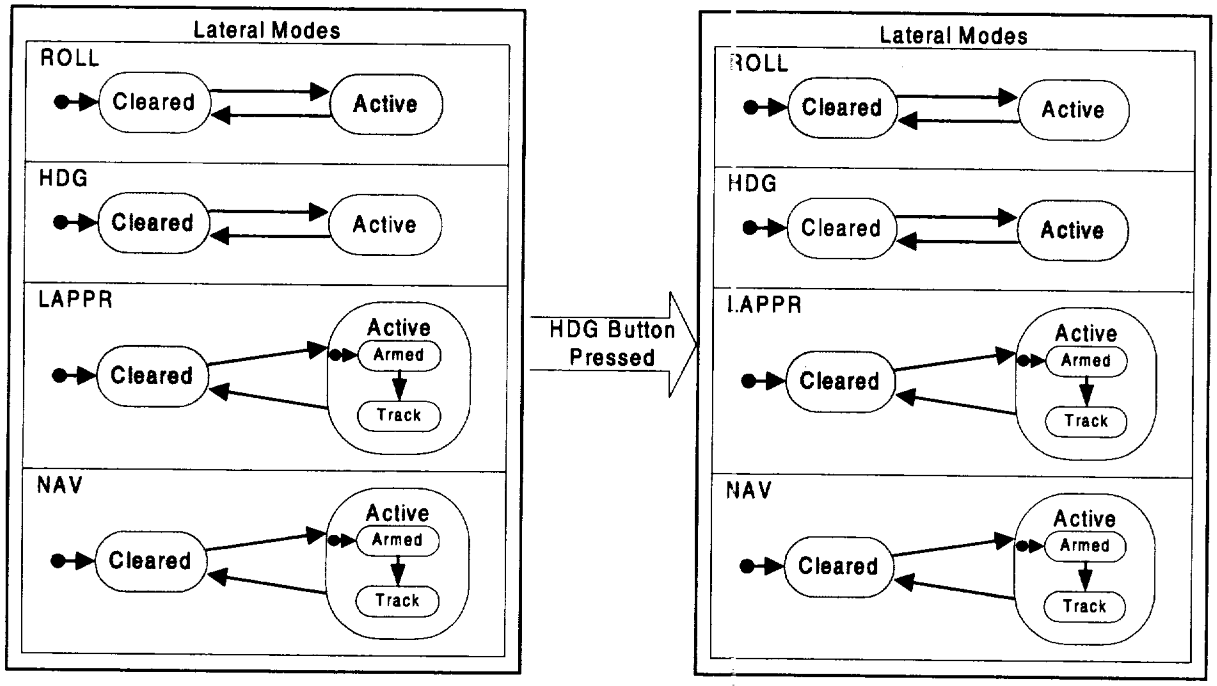
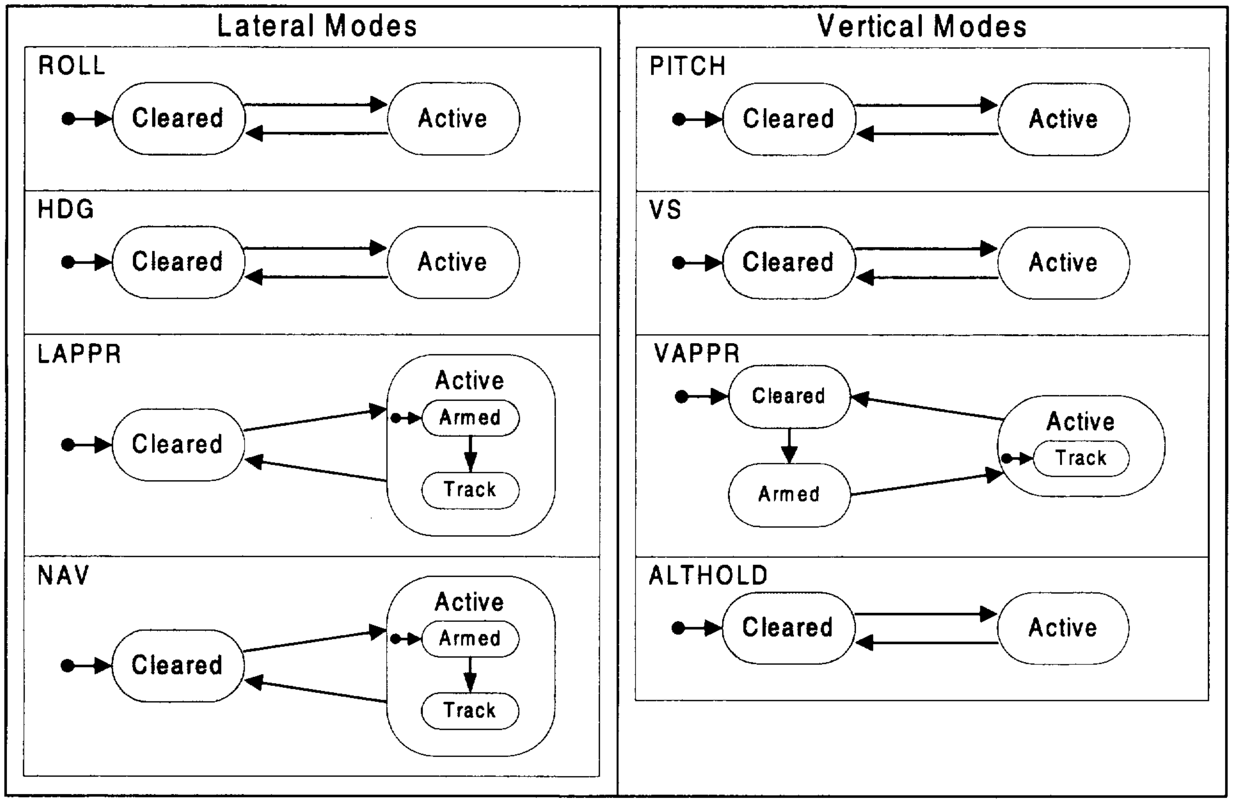


Figure 9 - Ensuring Only One Mode is Active

Although closely related, LAPPR and VAPPR behave quite differently. In LAPPR, the Armed state is an Active state, meaning that all other lateral modes are cleared when it is active. In the case of VAPPR, the Armed state is not an Active state. When VAPPR is armed, another vertical mode is actively producing pitch commands. In fact, the active vertical mode can be changed while VAPPR is armed. Trying to change the active lateral mode while LAPPR is armed clears LAPPR.

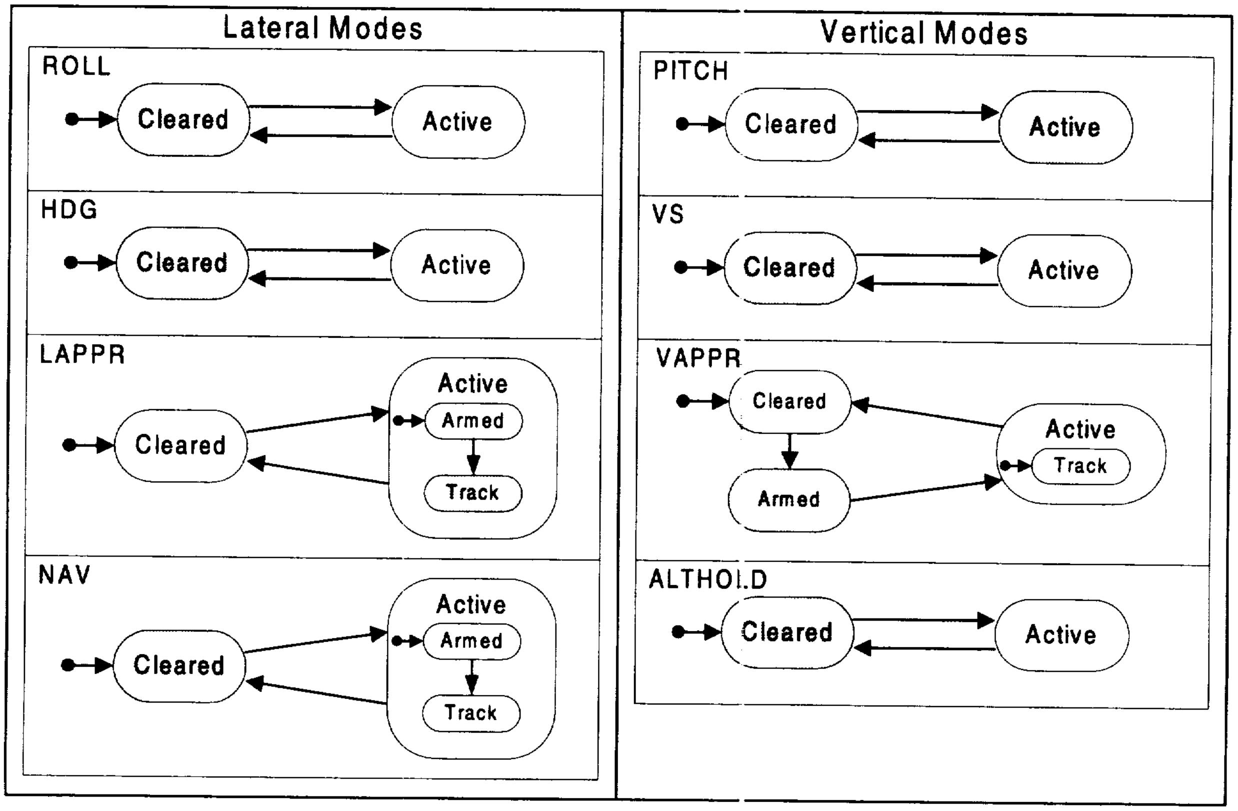
This difference is an example of a potential source of mode confusion cited by Leveson, et. al., [13], inconsistent behavior. While the difference is quite obvious in Figure 10 through Figure 12, it was obscure enough in the original CoRE specification [15] that it was missed in several inspections. This illustrates the value of creating a high level, executable model of the automation.

LAPPR and VAPPR could be changed to have similar behaviors simply by not making LAPPR Armed an active state. If desired, a constraint could be added such that whenever LAPPR entered the Armed state, HDG was forced to be the active lateral mode. Of course, whether this actually provided the desired behavior and reduced the potential for mode confusion would need to be determined by experts in human factors, pilots, and the system engineers.



Step 1 - LAPPR is Armed.

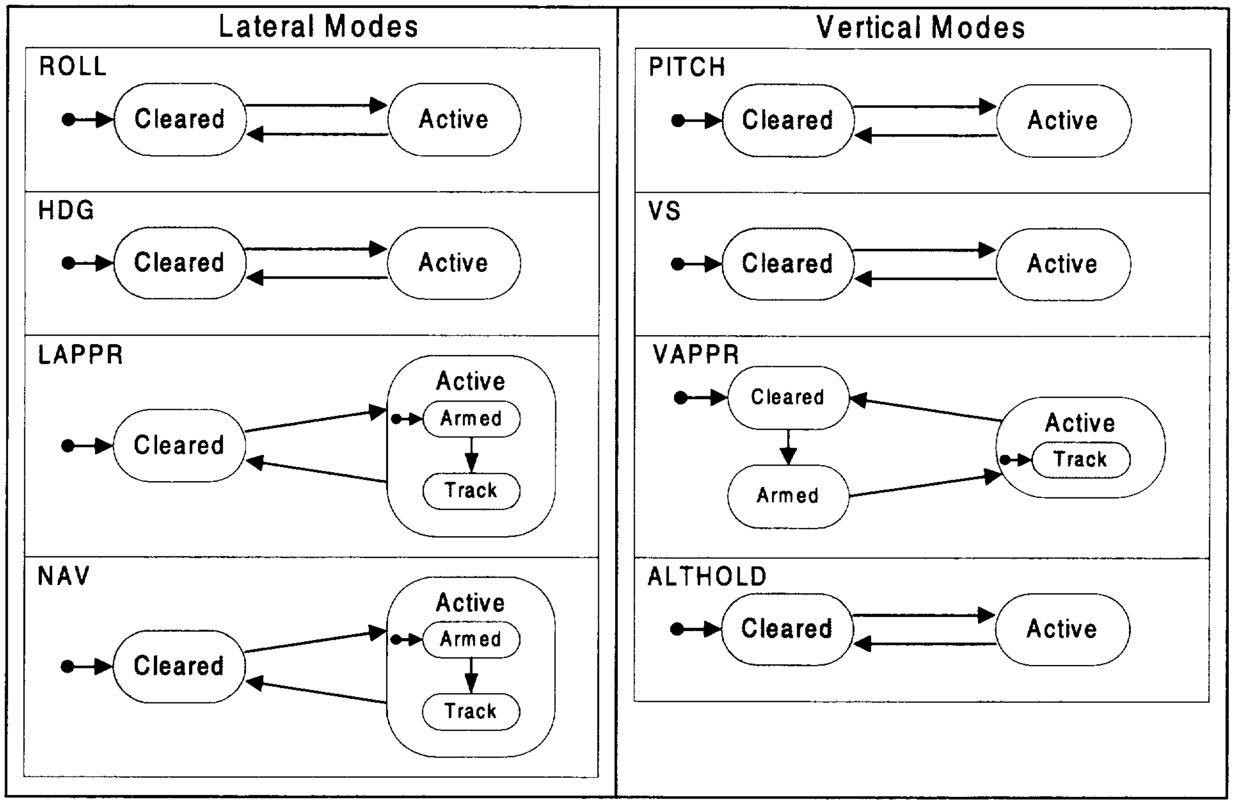
Figure 10 - Synchronization Between Lateral and Vertical Approach



Step 2 - The aircraft coming within lateral capture range causes LAPPR to change to Track.

Step 3 - LAPPR changing to Track causes VAPPR to change to armed.

Figure 11 - Synchronization Between Lateral and Vertical Approach (Continued)



Step 4 - The aircraft coming within vertical capture range causes VAPPR to change to Track, causing Pitch to clear.

Figure 12 - Synchronization Between Lateral and Vertical Approach (Continued)

4.3 Examples of Possible Sources of Mode Confusion

The ability to visualize both the cockpit and the Flight Guidance modes simultaneously is critical to understanding the actual behavior of the system. The following are examples of potential sources of mode confusion found in the example CoRE FGS specification [15] through visualization of the automation and cockpit.

4.3.1 Rotation of the VS/Pitch Wheel

In [13], Leveson et. al., identify inputs that are interpreted differently in different modes as a potential source of mode confusion. An example of this is the response of the system to a rotation of the VS/Pitch Wheel. When the active vertical mode is PITCH or VS, rotating the VS/Pitch Wheel on the flight control panel will not cause a mode change (see Figure 13). However, rotating the wheel when ALTHOLD is active will cause PITCH to become active, and ALTHOLD to clear (see Figure 14). Thus, in one mode, turning the VS/Pitch Wheel causes a mode change, while in another, it does not.

There are obvious reasons why rotating the VS/Pitch Wheel should not cause a mode change in PITCH or VS mode. While in PITCH mode, rotation of the wheel is used to change the pitch reference (desired pitch angle) of the aircraft. In VS mode, the wheel is used to change the vertical speed reference (desired vertical rate of ascent or descent). Using the same input device for both purposes conserves the limited space of the flight deck. Whether this is a significant source of mode confusion is a question that would need to be explored carefully by experts in human factors and flight deck design. In the same way, it can be argued whether rotating the VS/Pitch Wheel while in ALTHOLD mode should cause a mode change. It may be that this is an intuitive way for the pilot to interact with the system and is not a source of confusion.

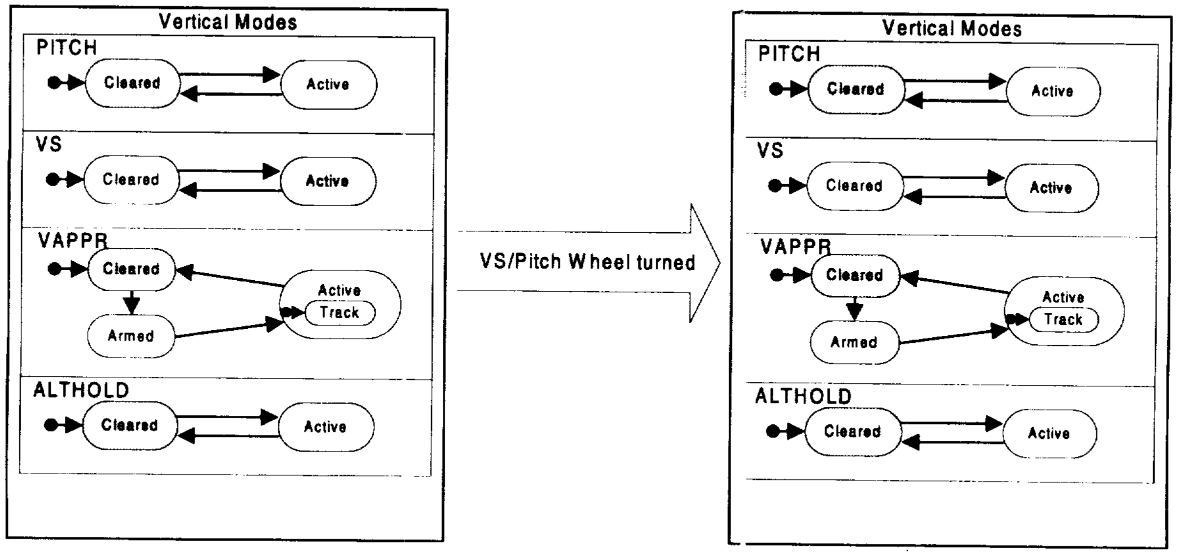


Figure 13 – Rotating the VS/Pitch Wheel Does Not Cause a Mode Change

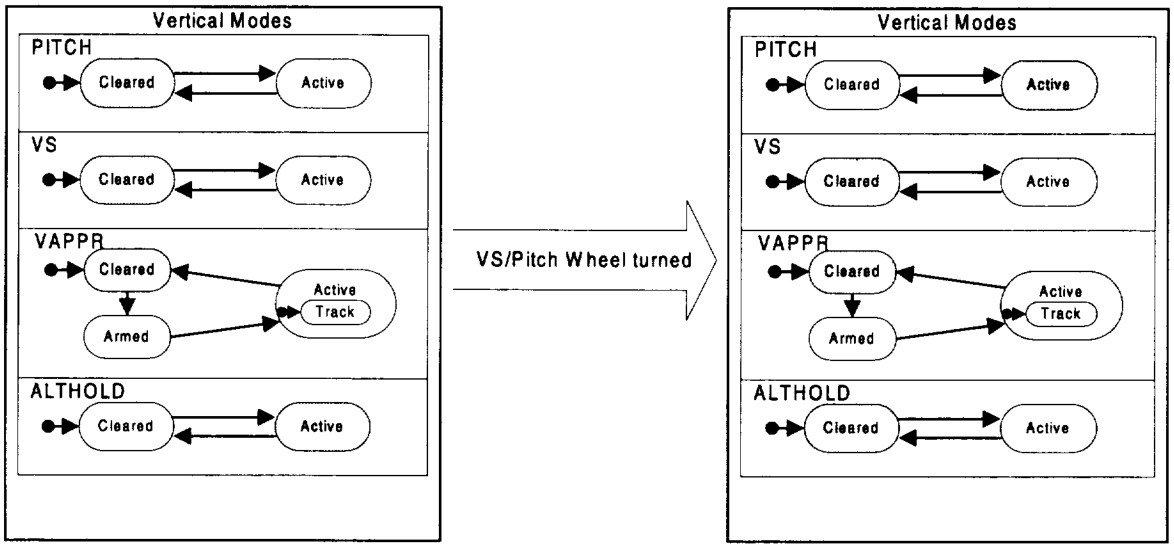


Figure 14 – Rotating the VS/Pitch Wheel Does Cause a Mode Change

4.3.2 Similar Annunciations

Another common source of mode confusion cited in [13] is lack of appropriate feedback. An example of this is the similarity of annunciations for NAV and APPR mode (see Figure 15 and Figure 16).

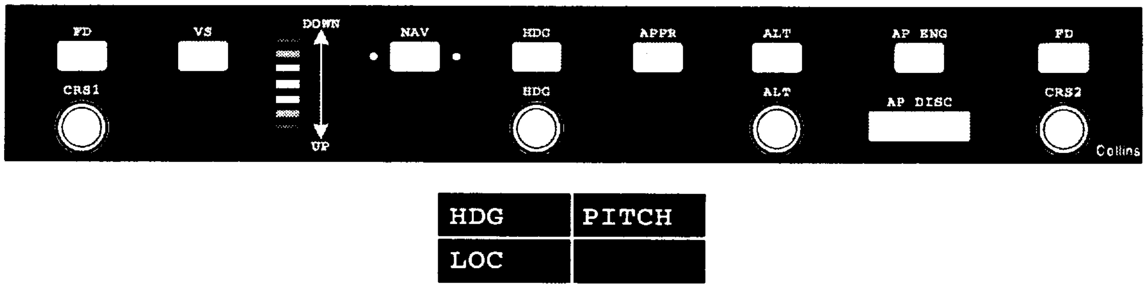


Figure 15 - NAV Mode Armed

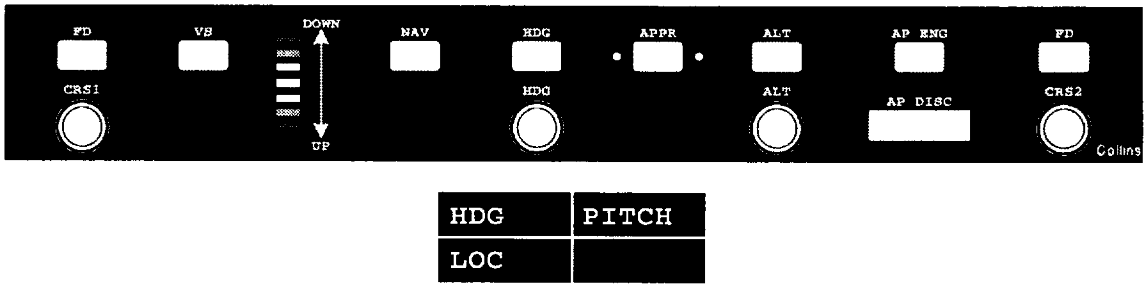


Figure 16 - APPR Mode Armed

The only difference in the CoRE FGS model in annunciating these two modes are the indicator lights around the NAV and APPR buttons. One way to resolve this would be to add a GS (glideslope) annunciation in the vertical armed field when lateral APPR mode is armed. The CoRE FGS model did not specify this since the vertical APPR mode does not enter the Armed state until the lateral APPR mode enters the Track state (see Section 4.2).

4.3.3 Response to the FD Switch

A final example of a potential source of mode confusion is the response to the FD button. The system interprets these differently depending on the current state of the system. When the Autopilot is not engaged and an overspeed condition does not exist, the FD switch will toggle the Flight Director between On and Off, removing both the mode annunciation and the guidance cues. When the Autopilot is engaged or an overspeed condition exists, the FD switch toggles between Cues-Off and Cues-On mode, removing the guidance cues but leaving the modes annunciated (see Figure 17).

Again, there are good reasons for this behavior. If the Autopilot is engaged, the Flight Director must be on since FAA regulations (and common sense) dictate that modes be annunciated when the Autopilot is engaged. For similar reasons, the modes are annunciated during an overspeed condition. Pressing the FD button while the Autopilot is engaged or an overspeed condition exists serves to both de-clutter the display and provide feedback to the pilot that the button press was recognized.

This example, as well as all those in this section, illustrate the need for potential sources of mode confusion to be carefully evaluated on a case by case basis. Often, there are constraints or conflicting requirements behind a particular design that are not readily apparent. Part of the contribution of this approach is to show how a high-level, executable model and a few inexpensive mock-ups can be used to detect potential sources of mode confusion and to facilitate discussion of how they can be resolved by the system designers, pilots, and experts in human factors.

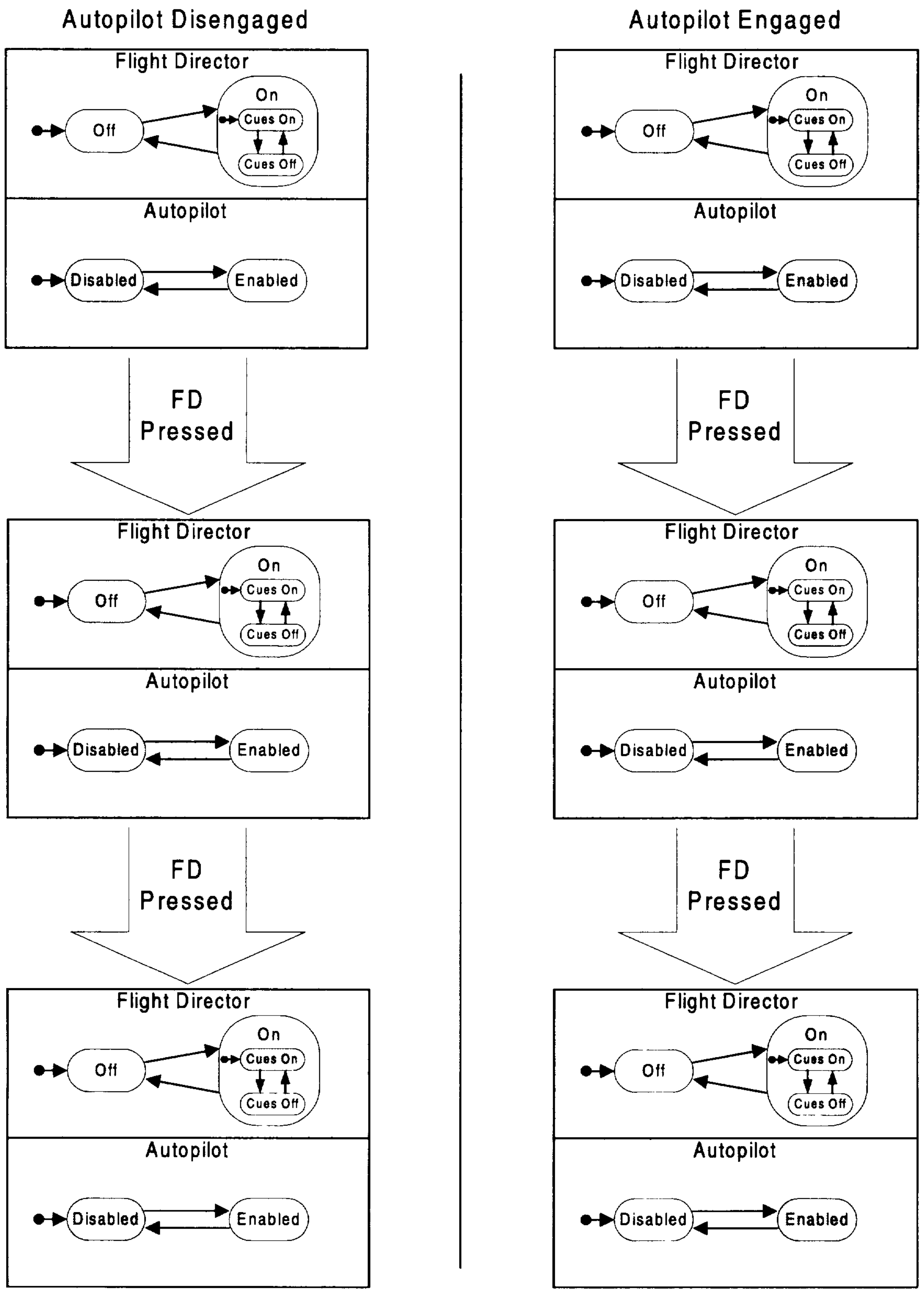


Figure 17 - Behavior of the FD Switch

Chapter 5
The Formal Model

The PVS specification of the mode logic follows the same architecture as was presented in Figure 6.[3] The import hierarchy for the PVS specification is shown in Figure 18. There is a theory `Flight_Guidance` that imports theories for `Flight_Director`, `Lateral_Guidance`, and `Vertical_Guidance`. `Lateral_Guidance` imports theories for the lateral modes and `Vertical_Guidance` imports theories for the vertical modes. However, many of the modes have identical state behavior and can be described by the same theory when only the mode logic is considered.[4] In the current FGS model, only three basic theories, `Simple_Guidance`, `Arming_Guidance`, and `Non-Arming_Guidance`, were needed to describe all the modes. The following sections describe the PVS specification starting with these three theories and building to `Flight_Guidance`. Names taken from the PVS theories are printed in italics.

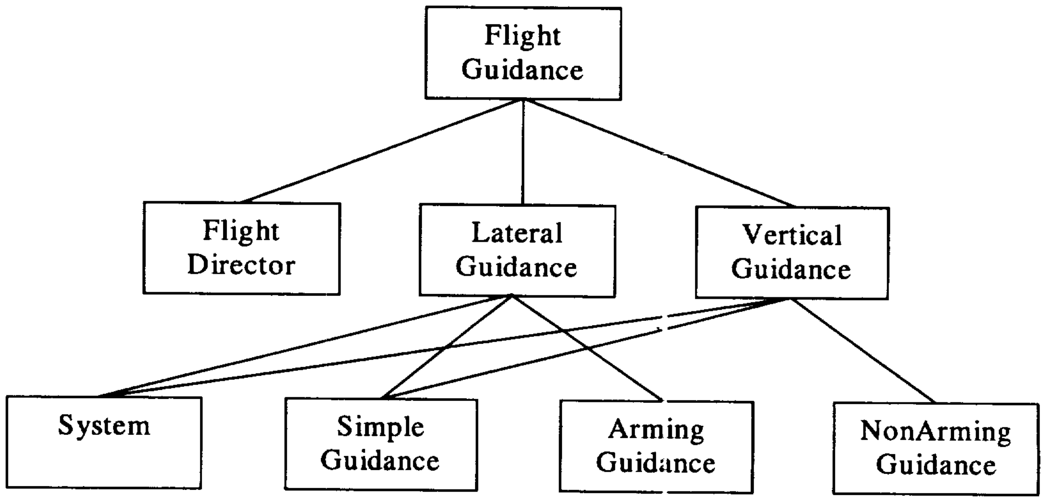


Figure 18 - PVS Import Hierarchy

[3] The PVS model completed in Phase I does not include all the modes shown in the visualization.
[4] If the this model were extended with additional information, e.g., the computation of the pitch or roll guidance command generated by the mode, this would no longer be true and separate theories would need to be defined for each mode. However, the common state behavior could still be shared among similar modes.

5.1 System Events

Events of interest to the entire Flight Guidance System, such as pressing the HDG switch, are defined in the theory `System` shown in Figure 19 and Figure 20. This theory enumerates all the system events and categorizes them according to which components of the Flight Guidance System they may affect. To facilitate reasoning about each mode separately and to use a template for defining similar modes, events are also defined locally within each component and a mapping is provided from the system events to these local events. Examples of this are provided in the following sections.

```
System: THEORY

  BEGIN

     %------------------------------------------------------------------------
     % Events seen by the FGS
     %------------------------------------------------------------------------
     Event: TYPE = { HDG_Switch_Hit,
                     NAV_Switch_Hit,
                     NAV_Armed_Long_Enough,
                     NAV_Track_Cond_Met_Event,
                     GA_Switch_Hit,
                     VS_Pitch_Wheel_Changed,
                     VS_Switch_Hit,
                     AP_Engaged,
                     AP_Disengaged,
                     FD_Switch_Hit,
                     Overspeed_Start,
                     Overspeed_End,
                     SYNC_Switch_Pressed,
                     SYNC_Switch_Released }

     %------------------------------------------------------------------------
     % There are no events that directly affect the lateral ROLL mode.
     % In this FGS, ROLL mode is the default lateral mode and is selected or
     % cleared by changing the other lateral modes.
     %------------------------------------------------------------------------
     ROLL_Event?(e:Event) : bool = False

     %------------------------------------------------------------------------
     % Events directly affecting the lateral HDG mode.
     %------------------------------------------------------------------------
     HDG_Event?(e:Event) : bool =  HDG_Switch_Hit?(e)

     %------------------------------------------------------------------------
     % Events directly affecting the lateral NAV mode.
     %------------------------------------------------------------------------
     NAV_Event?(e: Event) : bool =  NAV_Switch_Hit?(e) OR
         NAV_Armed_Long_Enough?(e) OR NAV_Track_Cond_Met_Event?(e)

     %------------------------------------------------------------------------
     % Events directly affecting the lateral GA mode.
     %------------------------------------------------------------------------
     LGA_Event?(e: Event) : bool =  GA_Switch_Hit?(e) OR
         AP_Engaged?(e) OR SYNC_Switch_Pressed?(e)

     %------------------------------------------------------------------------
     % Events directly affecting the active lateral mode.
     %------------------------------------------------------------------------
     Lateral_Event?(e: Event) : bool =
       ROLL_Event?(e) OR HDG_Event?(e) OR NAV_Event?(e) OR LGA_Event?(e)
```

Figure 19 - PVS Specification of System Events

5.2 State Behavior of the Operational Modes

The state behavior of the operational modes are described by three theories, Simple_Guidance, Arming_Guidance, and Non-Arming_Guidance. To provide a consistent interface to the Lateral_Guidance and Vertical_Guidance theories, all of these theories are structured similarly and provide similar responses to certain events. For example, the Activate event always takes a mode into an active state if it is in an inactive state, otherwise it is ignored. The Deactivate event always takes a mode into an inactive state if it is in an active state. The Switch event toggles a mode between active and inactive states, acting like an Activate event in an inactive mode and a Deactivate event in an active mode. Finally, the Clear event places the mode into the cleared state.

Each of the three theories contains a PVS record, State_Vector, that contains the current state of the mode and any additional state information associated with the mode. Since PVS does not provide explicit state variables, the state information for each mode is maintained in a State_Vector record in either Lateral_Guidance or Vertical_Guidance. The

```
    %-----------------------------------------------------------------------
    % Events directly affecting the vertical PITCH mode
    %-----------------------------------------------------------------------
    PITCH_Event?(e: Event) : bool = VS_Pitch_Wheel_Changed?(e)

    %-----------------------------------------------------------------------
    % Events directly affecting the vertical SPEED mode
    %-----------------------------------------------------------------------
    VS_Event?(e: Event)     : bool = VS_Switch_Hit?(e)

    %-----------------------------------------------------------------------
    % Events directly affecting the vertical GA mode.
    %-----------------------------------------------------------------------
    VGA_Event?(e: Event) : bool =  GA_Switch_Hit?(e) OR
        AP_Engaged?(e) OR SYNC_Switch_Pressed?(e)

    %-----------------------------------------------------------------------
    % Events directly affecting the active vertical mode.
    %-----------------------------------------------------------------------
    Vertical_Event?(e: Event) : bool =
      PITCH_Event?(e) OR VS_Event?(e) or VGA_Event?(e)

    %-----------------------------------------------------------------------
    % Lateral mode requests than can turn on the Flight Director
    %-----------------------------------------------------------------------
    Lateral_Mode_Requested?(e: Event) : bool =
      HDG_Switch_Hit?(e) OR NAV_Switch_Hit?(e) OR GA_Switch_Hit?(e)

    %-----------------------------------------------------------------------
    % Vertical mode requests that can turn on the Flight Director
    %-----------------------------------------------------------------------
    Vertical_Mode_Requested?(e: Event) : bool =
      VS_Switch_Hit?(e) OR GA_Switch_Hit?(e)

    %-----------------------------------------------------------------------
    % Events that affect the Flight Director
    %-----------------------------------------------------------------------
    Flight_Director_Event?(e: Event) : bool  =
         AP_Engaged?(e)                OR
         FD_Switch_Hit?(e)             OR
         Overspeed_Start?(e)           OR
         Lateral_Mode_Requested?(e)    OR
         Vertical_Mode_Requested?(e)

END System
```

Figure 20 - PVS Specification of System Events (Continued)

State_Vector records for Lateral_Guidance and Vertical_Guidance are in turn maintained in a State_Vector record in Flight_Guidance. While the intent is that the state information for a mode is encapsulated within the theory for that mode and is not directly manipulated by other theories, this is enforced only by convention.

Each mode theory also provides a function, next_state, that takes the current state of a mode and an event and returns the next state and a set of internal events, or signals, that need to be acted on by other components. For example, if a mode becomes active, the next_state function returns both the new state and the Activated signal. This signal is used by either Lateral_Guidance or Vertical_Guidance to ensure that the current active mode is cleared. Finally, each theory provides a predicate Active? that takes the current state of the mode and returns true if the mode is active.

5.2.1 Simple Guidance

The state behavior for Simple_Guidance is depicted in Figure 21. As its name implies, modes exhibiting this behavior only have two states, CLEARED and ACTIVE, and a handful of transitions between them. Since these modes are so simple, the Clear and Deactivate events cause identical state changes. Both events are defined in order to provide an interface consistent with those of other modes.

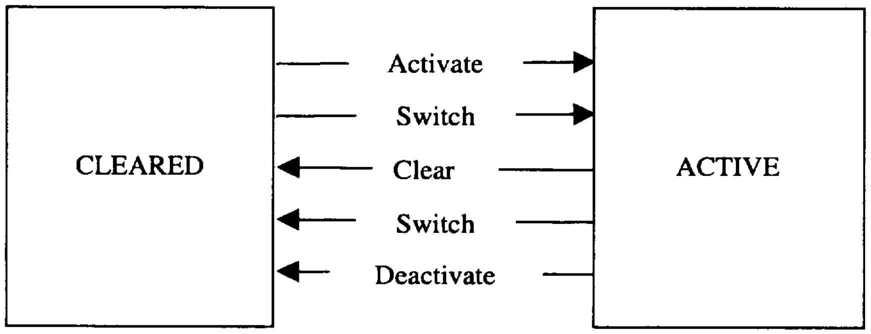


Figure 21 - State Behavior for Simple Guidance

The PVS specification Simple_Guidance is shown in Figure 22. The two states for the mode are defined as a PVS enumeration type. For modes that can be described by Simple_Guidance, the State_Vector record only contains the current state of the mode. The mode is Active? if the value of this field is ACTIVE.

To facilitate using Simple_Guidance as a template for several modes, the events that cause state transitions in Simple_Guidance are listed here as the PVS enumeration type Event. The external system events listed in Figure 19 and Figure 20 are mapped into these events in the Lateral_Guidance and Vertical_Guidance theories. The internal events, or signals, that may be raised in Simple_Guidance are defined as the enumeration type Signal. The next_state function defines how a Simple_Guidance mode responds to each event by returning the new state and the set of signals raised for synchronization with the other modes. If no further synchronization is necessary, a single Null? signal is returned.

```
Simple_Guidance: THEORY

BEGIN

    %-----------------------------------------------------------------------
    % Mode States
    %-----------------------------------------------------------------------
    State:  TYPE = {CLEARED, ACTIVE}

    %-----------------------------------------------------------------------
    % State vector for a Simple Guidance Mode
    %-----------------------------------------------------------------------
    State_Vector: TYPE = [# state: State #]

    %-----------------------------------------------------------------------
    % Useful Definitions
    %-----------------------------------------------------------------------
    Active?(s: State_Vector): bool  = ACTIVE?(state(s))

    %-----------------------------------------------------------------------
    % Incoming Events and Outgoing Signals
    %-----------------------------------------------------------------------
    Event:  TYPE = {Activate, Switch, Clear, Deactivate}

    Signal: TYPE = {Null, Activated, Deactivated}

    %-----------------------------------------------------------------------
    % Next state function
    %-----------------------------------------------------------------------

    next_state(s: State_Vector, e: Event): [State_Vector, set[Signal] ] =
      IF CLEARED?(state(s)) THEN
         COND
           Activate?(e)    ->  ((# state := ACTIVE # , Activated?),
           Switch?(e)      ->  ((# state := ACTIVE # , Activated?),
           Clear?(e)       ->  (          s,           Null?),
           Deactivate?(e)  ->  (          s,           Null?)
         ENDCOND
      ELSE % IF ACTIVE?(s) THEN
         COND
           Deactivate?(e)  -> ((# state := CLEARED # , Deactivated?),
           Switch?(e)      -> ((# state := CLEARED # , Deactivated?),
           Clear?(e)       -> ((# state := CLEARED # , Deactivated?),
           Activate?(e)    -> (         s,             Null?)
         ENDCOND
      ENDIF

END Simple_Guidance
```

Figure 22 - PVS Specification of Simple Guidance

5.2.2 Arming Guidance

The state behavior for Arming_Guidance is depicted in Figure 23. Arming_Guidance extends the behavior of Simple_Guidance by adding sub-states ARMED and TRACK of the ACTIVE state, and sub-states ARMED_INITIAL and ARMED_LONG_ENOUGH of the ARMED state. Arming_Guidance modes are those in which the system can be armed pending the capture of a navigation source, but is also active and generating guidance commands while armed. For example, the lateral navigation (LNAV) mode can generate lateral guidance commands to fly a specific heading while armed for the capture of a navigation source such as a VOR. In TRACK mode, the navigation source has been captured and the mode is generating guidance commands to track that source. The sub-states ARMED_INITIAL and ARMED_LONG_ENOUGH are used to ensure that the system remains in the ARMED mode a minimum period of time even if the conditions for capturing the navigation source are satisfied on entry to the ARMED mode.

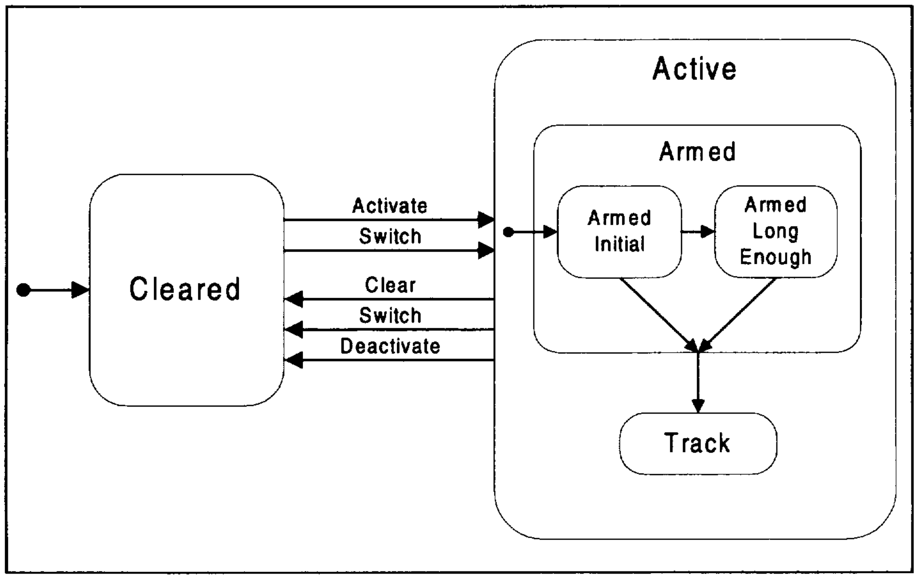


Figure 23 - State Behavior for Arming Guidance

The PVS specification for Arming_Guidance is given in Figure 24. The enumeration of states is extended to include ARMED_INITIAL, ARMED_LONG_ENOUGH, and TRACK. The ARMED and ACTIVE states are defined as predicates over these base states. The State_Vector contains the current state of the mode and a boolean, Track_Cond_Met?, that indicates whether the conditions for capture of the navigation source are met. The list of events are extended to include the event Track_Cond_Met that occurs when the condition for capture of the navigation source are met and the event Armed_Long_Enough that occurs when the state has been ARMED for the minimum acceptable amount of time. Finally, the next_state function defines the new state created and signals raised by processing an event.

```
Arming_Guidance: THEORY
  BEGIN

    %-------------------------------------------------------------------
    % Mode States
    %-------------------------------------------------------------------
    State:  TYPE = {CLEARED, ARMED_INITIAL, ARMED_LONG_ENOUGH, TRACK}

    ARMED?(s: State) : bool = ARMED_INITIAL?(s) OR ARMED_LONG_ENOUGH?(s)
    ACTIVE?(s: State): bool = ARMED?(s) OR TRACK?(s)

    %-------------------------------------------------------------------
    % State vector
    %-------------------------------------------------------------------
    State_Vector: TYPE = [# state: State,
                            Track_Cond_Met? : bool #]

    %-------------------------------------------------------------------
    % Useful Definitions
    %-------------------------------------------------------------------
    Active?(s: State_Vector) : bool  = ACTIVE?(state(s))

    %-------------------------------------------------------------------
    % Incoming Events and Outgoing Signals
    %-------------------------------------------------------------------
    Event:  TYPE = {Activate, Clear, Switch, Deactivate,
                    Track_Cond_Met, Armed_Long_Enough}

    Signal: TYPE = {Null, Activated, Deactivated}

    %-------------------------------------------------------------------
    % Next state function
    %-------------------------------------------------------------------
    next_state(s: State_Vector, e: Event ): [State_Vector, set[Signal] ] =
     IF CLEARED?(state(s)) THEN
      COND
        Activate?(e)          -> (s WITH [state := ARMED_INITIAL], Activated?),
        Switch?(e)            -> (s WITH [state := ARMED_INITIAL], Activated?),
        Clear?(e)             -> (            s,                 Null?),
        Deactivate?(e)        -> (            s,                 Null?),
        Track_Cond_Met?(e)    -> (s WITH [Track_Cond_Met? := true],Null?),
        Armed_Long_Enough?(e) -> (            s,                 Null?)
      ENDCOND
    ELSE % IF ACTIVE?(state(s))
      COND
        Activate?(e)          -> (            s,                 Null?),
        Switch?(e)            -> (s WITH [state := CLEARED],      Deactivated?),
        Clear?(e)             -> (s WITH [state := CLEARED],      Deactivated?),
        Deactivate?(e)        -> (s WITH [state := CLEARED],      Deactivated?),
        Track_Cond_Met?(e)    ->
          IF ARMED_LONG_ENOUGH?(state(s))
             THEN                (s WITH [state := TRACK,
                                          Track_Cond_Met? := true],Null?)
             ELSE                (s WITH [Track_Cond_Met? := true],Null?)
          ENDIF,
        Armed_Long_Enough?(e) ->
          IF ARMED_INITIAL?(state(s)) & Track_Cond_Met?(s)
             THEN                (s WITH [state := TRACK],        Null?)
          ELSIF ARMED_INITIAL?(state(s)) & NOT Track_Cond_Met?(s)
             THEN                (s WITH [state := ARMED_LONG_ENOUGH], Null?)
          ELSE                   (           s,                   Null?)
          ENDIF
       ENDCOND
     ENDIF

END Arming_Guidance
```

Figure 24 - PVS Specification of Arming Guidance

5.3 Lateral Guidance

As discussed in Section 4.1, Lateral_Guidance contains the lateral modes and maintains any constraints between them. The PVS specification of Lateral_Guidance is shown in Figure 25 and Figure 26. The specification of the lateral modes of ROLL, HDG, and GA are created by importing the `Simple_Guidance` theory and assigning each mode an abbreviation. The specification of lateral `NAV` mode is created by importing the `Arming_Guidance` theory with an abbreviation. The `State_Vector` for `Lateral_Guidance` consists of the state vectors for each of these modes.

The `Clear_All_Modes` function is used by `Flight_Guidance` to clear (turn off) all lateral modes when the Flight Director is turned off. It uses the `next_state` function of each mode to keep the PVS specification as close as possible to the ObjecTime model used for visualization.

```
Lateral_Guidance : THEORY

  BEGIN

    IMPORTING System

    ROLL: THEORY = Simple_Guidance
    HDG:  THEORY = Simple_Guidance
    NAV:  THEORY = Arming_Guidance
    GA:   THEORY = Simple_Guidance

    State_Vector: TYPE = [# ROLL: ROLL.State_Vector,
                            HDG:  HDG.State_Vector,
                            NAV:  NAV.State_Vector,
                            GA:   GA.State_Vector    #]

    %-----------------------------------------------------------------------
    % Clear all lateral modes
    %-----------------------------------------------------------------------
    Clear_All_Modes(s: State_Vector): State_Vector =
       (# ROLL := proj_1(next_state(ROLL(s), Clear)),
          HDG  := proj_1(next_state(HDG(s),  Clear)),
          NAV  := proj_1(next_state(NAV(s),  Clear)),
          GA   := proj_1(next_state(GA(s),   Clear))  #)

    %-----------------------------------------------------------------------
    % Deactivate all lateral modes.  Note that this function changes a mode
    % only if it is active.
    %-----------------------------------------------------------------------
    Deactivate_All_Modes(s: State_Vector): State_Vector =
       (# ROLL := proj_1(next_state(ROLL(s), Deactivate)),
          HDG  := proj_1(next_state(HDG(s),  Deactivate)),
          NAV  := proj_1(next_state(NAV(s),  Deactivate)),
          GA   := proj_1(next_state(GA(s),   Deactivate)) #)

    %-----------------------------------------------------------------------
    % Select (activate) the default mode
    %-----------------------------------------------------------------------
    Select_Default_Mode(s: State_Vector): State_Vector =
      s WITH [ ROLL := proj_1(next_state(ROLL(s), Activate)) ]

    %-----------------------------------------------------------------------
    % Map system HDG events onto the events of HDG theory
    %-----------------------------------------------------------------------
    HDG_Event(e: (HDG_Event?)) : HDG.Event =
       COND
          HDG_Switch_Hit?(e) -> HDG.Switch
       ENDCOND

    %-----------------------------------------------------------------------
    % Map system NAV events onto the events of NAV theory
    %-----------------------------------------------------------------------
    NAV_Event(e: (NAV_Event?)) : NAV.Event =
       COND
         NAV_Switch_Hit?(e)           -> NAV.Switch,
         NAV_Track_Cond_Met_Event?(e) -> NAV.Track_Cond_Met,
         NAV_Armed_Long_Enough?(e)    -> NAV.Armed_Long_Enough
       ENDCOND
```

Figure 25 - PVS Specification of Lateral Guidance

The Deactivate_All_Modes function is used by Lateral_Guidance to ensure that at most one lateral mode is ever active. This function changes the state of a mode only if the mode is active. The Select_Default_Mode function is used by Lateral_Guidance to activate the default mode when the current active lateral mode is deactivated and by Flight_Guidance to activate the default mode when the Flight_Director is turned on without the selection of a specific lateral mode.

```
%------------------------------------------------------------------------
% Map system LGA events onto the events of GA theory
%------------------------------------------------------------------------
GA_Event(e: (LGA_Event?)) : GA.Event =
   COND
     GA_Switch_Hit?(e)             -> GA.Activate,
     AP_Engaged?(e)                -> GA.Clear,
     SYNC_Switch_Pressed?(e)       -> GA.Clear
   ENDCOND

Signal: TYPE = {Null}

%------------------------------------------------------------------------
% Process the next event
%------------------------------------------------------------------------
next_state(s:State_Vector, e:System.Event): [State_Vector, set[Signal] ] =
 COND

  %-----------------------------------------------------------------------
  HDG_Event?(e) ->
  %-----------------------------------------------------------------------
  LET (newHDG, signals) = next_state(HDG(s), HDG_Event(e))
  IN IF signals(Activated) THEN
        (Deactivate_All_Modes(s) WITH [HDG := newHDG], Null?)
     ELSIF signals(Deactivated) THEN
        (Select_Default_Mode(s  WITH [HDG := newHDG]), Null?)
     ELSE
        (s  WITH [HDG := newHDG], Null?)
  ENDIF,

  %-----------------------------------------------------------------------
  NAV_Event?(e) ->
  %-----------------------------------------------------------------------
  Let (newNAV, signals) = next_state(NAV(s), NAV_Event(e))
  IN IF signals(Activated) THEN
        (Deactivate_All_Modes(s) WITH [NAV := newNAV], Null?)
     ELSIF signals(Deactivated) THEN
        (Select_Default_Mode(s WITH [NAV := newNAV]),  Null?)
     ELSE
        (s  WITH [NAV := newNAV], Null?)
     ENDIF,

  %-----------------------------------------------------------------------
  LGA_Event?(e) ->
  %-----------------------------------------------------------------------
  LET (newGA, signals) = next_state(GA(s), GA_Event(e))
  IN IF signals(Activated) THEN
        (Deactivate_All_Modes(s) WITH [GA := newGA], Null?)
     ELSIF signals(Deactivated) THEN
        (Select_Default_Mode(s WITH [GA := newGA ]), Null?)
     ELSE
        (s  WITH [GA := newGA], Null?)
     ENDIF,

  ELSE -> (s, Null?)

 ENDCOND

END Lateral_Guidance
```

Figure 26 - PVS Specification of Lateral Guidance (Continued)

The system level events defined in theory System (Figure 19 and Figure 20) are mapped into the internal events of Simple_Guidance and Arming_Guidance by the functions HDG_Event, NAV_Event, and GA_Event. For example, GA_Event maps the system event AP_Engaged into the Simple_Guidance event Clear for theory GA.

The next_state function for Lateral_Guidance takes a state vector for Lateral_Guidance and a system event and returns the new state vector and a set of signals raised. Since the system events have been categorized by the modes of the FGS they affect, the next_state function first determines if the event could affect a particular mode, then computes the new state of the mode, then synchronizes any changes in the state of the mode with the other lateral modes. For example, if the event is a HDG_Event, the next_state function first computes the next state of the HDG component (recall that the Simple_Guidance theory used to create the HDG theory also has a next_state function). If the Activate signal was raised, indicating that the HDG mode has become active as a result of the event, then all modes in the lateral guidance state vector are deactivated and the new active HDG mode is installed in the state vector. In like fashion, if the Deactivate signal was raised, indicating that the HDG mode has become inactive, then the new inactive HDG mode is installed in the state vector and the default mode is selected (activated). If no signals were raised, no synchronization with the other modes are necessary and the new HDG mode is simply installed in the Lateral_Guidance state vector. Finally, note that if the event does not affect any lateral modes, the original state vector and the null signal are returned.

5.4 Vertical Guidance

Vertical_Guidance plays the same role for the vertical modes that Lateral_Guidance does for the lateral modes. The PVS specification for Vertical_Guidance is shown in Figure 27 and Figure 28. The specification of the vertical modes of PITCH, VS, and GA are created by importing the Simple_Guidance theory and assigning each mode an abbreviation. The State_Vector for Vertical_Guidance consists of the state vectors for each of these modes.

The Clear_All_Modes, Deactivate_All_Modes, and Select_Default_Mode functions serve the same functions as in Lateral_Guidance, but for the vertical modes. The default vertical mode is PITCH. The next_state function is also similar to that for Lateral_Guidance

```
Vertical_Guidance: THEORY
  BEGIN

    IMPORTING System

    PITCH: THEORY = Simple_Guidance
    VS:    THEORY = Simple_Guidance
    GA:    THEORY = Simple_Guidance

    State_Vector: TYPE = [# PITCH : PITCH.State_Vector,
                            VS    : VS.State_Vector,
                            GA    : GA.State_Vector      #]

    %-----------------------------------------------------------------------------
    % Clear all vertical modes.
    %-----------------------------------------------------------------------------
    Clear_All_Modes(s: State_Vector): State_Vector =
       (# PITCH := proj_1(next_state(PITCH(s), Clear)),
          VS    := proj_1(next_state(VS(s), Clear)),
          GA    := proj_1(next_state(VS(s), Clear)) #)

    %-----------------------------------------------------------------------------
    % Deactivate all vertical modes.  Note that this function changes a mode
    % only if it is active.
    %-----------------------------------------------------------------------------
    Deactivate_All_Modes(s: State_Vector): State_Vector =
       (# PITCH := proj_1(next_state(PITCH(s), Deactivate)),
          VS    := proj_1(next_state(VS(s), Deactivate)),
          GA    := proj_1(next_state(GA(s), Deactivate)) #)

    %-----------------------------------------------------------------------------
    % Select (activate) the default mode
    %-----------------------------------------------------------------------------
    Select_Default_Mode(s: State_Vector): State_Vector =
      s WITH [ PITCH := proj_1(next_state(PITCH(s), Activate)) ]

    %-----------------------------------------------------------------------------
    % Map system PITCH events onto the events of PITCH theory
    %-----------------------------------------------------------------------------
    PITCH_Event(e: (PITCH_Event?)) : PITCH.Event =
       COND
         VS_Pitch_Wheel_Changed?(e) -> PITCH.Activate
       ENDCOND

    %-----------------------------------------------------------------------------
    % Map system VS events onto the events of VS theory
    %-----------------------------------------------------------------------------
    VS_Event(e: (VS_Event?)) : VS.Event =
       COND
         VS_Switch_Hit?(e) -> VS.Switch
       ENDCOND

    %-----------------------------------------------------------------------------
    % Map system VGA events onto the events of GA theory
    %-----------------------------------------------------------------------------
    GA_Event(e: (VGA_Event?)) : GA.Event =
       COND
         GA_Switch_Hit?(e)          -> GA.Switch,
         AP_Engaged?(e)             -> GA.Clear,
         SYNC_Switch_Pressed?(e)    -> GA.Clear
       ENDCOND
```

Figure 27 - PVS Specification of Vertical Guidance

```
  Signal: TYPE = {Null}

  %--------------------------------------------------------------------------
  % Process the next event
  %-------------------------------------------------------------------------
  next_state(s: State_Vector, e: System.Event):
     [State_Vector, set[Signal] ] =
  COND

    %------------------------------------------------------------------------
    PITCH_Event?(e) ->
    %------------------------------------------------------------------------
    LET (newPITCH, signals) = next_state(PITCH(s), PITCH_Event(e))
    IN IF signals(Activated) THEN
          (Deactivate_All_Modes(s) WITH [PITCH := newPITCH], Null?)
       ELSIF signals(Deactivated) THEN
          (Select_Default_Mode(s WITH [PITCH := newPITCH] ), Null?)
       ELSE
          (s WITH [PITCH := newPITCH], Null?)
       ENDIF,

    %------------------------------------------------------------------------
    VS_Event?(e) ->
    %------------------------------------------------------------------------
    LET (newVS, signals) = next_state(VS(s), VS_Event(e))
    IN IF signals(Activated) THEN
         (Deactivate_All_Modes(s) WITH [VS := newVS], Null?)
       ELSIF signals(Deactivated) THEN
         (Select_Default_Mode(s WITH [VS := newVS]), Null?)
       ELSE
         (s WITH [VS := newVS], Null?)
       ENDIF,

    %------------------------------------------------------------------------
    VGA_Event?(e) ->
    %------------------------------------------------------------------------
    LET (newGA, signals) = next_state(GA(s), GA_Event(e))
    IN IF signals(Activated) THEN
          (Deactivate_All_Modes(s) WITH [GA := newGA], Null?)
       ELSIF signals(Deactivated) THEN
          (Select_Default_Mode(s WITH [GA := newGA]), Null?)
       ELSE
          (s WITH [GA := newGA ], Null?)
       ENDIF,

    ELSE -> (s, Null?)

  ENDCOND

END Vertical_Guidance
```

Figure 28 - PVS Specification of Vertical Guidance (Continued)

5.5 Flight Director

The Flight Director defines when the active and armed modes are annunciated and when the guidance cues are displayed on the EFIS display. The state behavior of the Flight Director is shown in Figure 29. The Flight Director has two main states, OFF and ON. When in the OFF state, the active and armed lateral and vertical modes are not annunciated and the guidance cues are not displayed on the EFIS. When in the ON state, the active and armed lateral and vertical modes are always annunciated on the EFIS. The ON state has two sub-states, CUES OFF and CUES ON, that determine if the guidance cues are displayed or not.

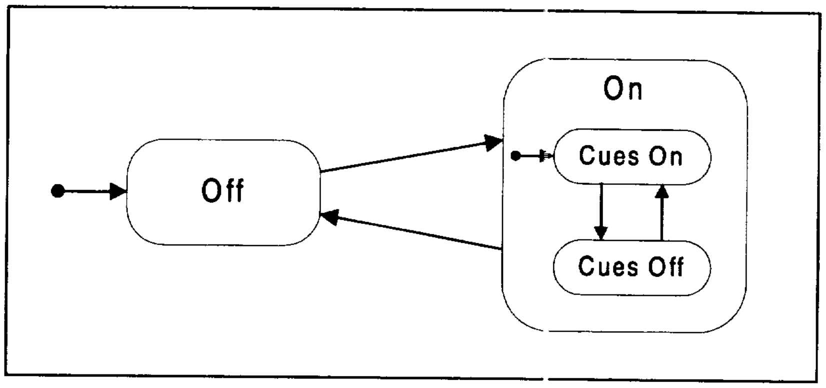


Figure 29 - State Behavior for Flight Director

The PVS specification for the `Flight_Director` is shown in Figure 30. Events local to the `Flight_Director` are those to `Turn_On`, `Turn_Off`, `Force_Cues` to force the guidance cues to display, and `Switch` to toggle the `Flight_Director` on and off. The `Flight_Director` returns two signals, `Turned_On` and `Turned_Off`, used to coordinate with the rest of the Flight Guidance System. The `State_Vector` consists simply of the Flight Director's state. For convenience in the `Flight_Guidance` theory, the function `On?` is defined here as an abbreviation.

The `next_state` function for the `Flight_Director` differs from the next state functions encountered so far in that they include two additional boolean parameters, `AP_Engaged?` and `Overspeed?`. These are components of the Flight Guidance System state external to the `Flight_Director` that affect its behavior. `AP_Engaged?` Indicates whether the Autopilot is engaged. `Overspeed?` indicates if the airspeed of the aircraft exceeds its structural limits. These are used as guards on some of the `Flight_Director` state transitions. In particular, if the autopilot is engaged or the overspeed condition exists, the `Flight_Director` must be in the ON state so that the lateral and vertical modes are annunciated.

```
Flight_Director: THEORY

BEGIN

    %------------------------------------------------------------------------
    % States
    %------------------------------------------------------------------------
    State:  TYPE = {OFF, CUES, NO_CUES}

    ON?(s: State) : bool = CUES?(s) OR NO_CUES?(s)

    %------------------------------------------------------------------------
    % Events and Signals
    %------------------------------------------------------------------------
    Event:  TYPE = {Turn_On, Force_Cues, Switch, Turn_Off }

    Signal: TYPE = {Null, Turned_On, Turned_Off}

    %------------------------------------------------------------------------
    % State vector for the Flight Director
    %------------------------------------------------------------------------
    State_Vector: TYPE = [# state       : State #]

    %------------------------------------------------------------------------
    % Useful definition
    %------------------------------------------------------------------------
    On?(s: State_Vector): bool = ON?(state(s))

    %------------------------------------------------------------------------
    % Next state function
    %------------------------------------------------------------------------

    next_state(s:State_Vector, e:Event, AP_Engaged?:bool, Overspeed?:bool):
       [State_Vector, set[Signal]] =

    IF OFF?(state(s)) THEN
       COND
         Force_Cues?(e)         -> (s WITH [state := CUES],  Turned_On?),
         Turn_On?(e)            -> (s WITH [state := CUES],  Turned_On?),
         Switch?(e)             -> (s WITH [state := CUES],  Turned_On?),
         Turn_Off?(e)           -> (              s,         Null?)
       ENDCOND
    ELSIF CUES?(state(s)) THEN
       COND
         Force_Cues?(e)         -> (              s,         Null?),
         Turn_On?(e)            -> (              s,         Null?),
         Switch?(e) OR Turn_Off?(e) ->
            IF (Overspeed? or AP_Engaged?) THEN
                                   (s WITH [state := NO_CUES], Null?)
            ELSE
                                   (s WITH [state := OFF],   Turned_Off?)
            ENDIF
         ENDCOND
    ELSE % IF NO_CUES(state(s)) THEN
        COND
          Force_Cues?(e)        -> (s WITH [state := CUES],  Null?),
          Turn_On?(e)           -> (s WITH [state := CUES],  Null?),
          Switch?(e)            ->
            IF (Overspeed? or AP_Engaged?) THEN
                                   (s WITH [state := CUES],  Null?)
            ELSE
                                   (s WITH [state := OFF],   Turned_Off?)
            ENDIF,
          Turn_Off?(e)          ->
            IF (Overspeed? or AP_Engaged?) THEN
                                   (              s,          Null?)
            ELSE
                                   (s WITH [state := OFF],    Turned_Off?)
            ENDIF
         ENDCOND
    ENDIF

END Flight_Director
```

Figure 30 - PVS Specification of Flight Director

5.6 Flight Guidance

Flight_Guidance contains the Flight_Director, Lateral_Guidance, and Vertical_Guidance components. The PVS specification for this theory is shown in Figure 31. The State_Vector for the entire Flight_Guidance system consists of the State_ Vectors for these components and the AP_Enaged? And Overspeed? indicators provided by sources external to the Flight Guidance System. Just as was done in Lateral_Guidance and Vertical_Guidance, the FD_Event function maps the system level events that can affect the Flight_Director into the local events for that theory.

The next_state function for the entire Flight Guidance System is defined in terms of three auxiliary functions, Process_External_Event, Process_FD_Event, and Process_ Flight_Mode_Event. Each of these take a system event and a State_Vector and return a new State_Vector.

The Process_External_Event is the simplest of the three. It simply updates AP_Engaged? and Overspeed? booleans if the system event indicates their status has changed.

The Process_FD_Event function updates the Flight_Director component of the system state if the event might affect it. It first determines the new state of the Flight_ Director. If the Flight_Director was Turned_Off, then the lateral and vertical modes are cleared and the existing state is updated with the new state of the Flight_Director and the cleared lateral and vertical modes. If the Flight_Director was Turned_On, then default lateral and vertical modes are selected and the existing state is updated with the new state of the Flight_Director and the lateral and vertical modes. If the Flight_Director was neither Turned_On or Turned_Off by the event (for example, if the guidance cues were simply turned off), then the existing state is updated with the new state of the Flight_ Director.

The Process_Flight_Modes_Event determines any changes caused in the lateral and vertical modes by the event and installs the new modes in the State_Vector. No change is made if the Flight_Director is turned off.

The next_state function takes a State_Vector and a system event and returns the next_state of Flight_Guidance. No signals are returned from the next_state function as Flight_Guidance is the top most theory in the specification and there are no sibling components to maintain synchronization with. It first computes any changes in the AP_Engaged? and Overspeed? booleans, then computes any changes to the Flight_ Director, and finally computes any changes in the lateral and vertical modes. This tiered strategy is necessary since an event may affect multiple components. For example, if the Flight Director is off, pressing the HDG button will turn the Flight_Director on and select the HDG lateral mode and the default vertical mode.

```
Flight_Guidance : THEORY

  BEGIN

    IMPORTING Flight_Director, Lateral_Guidance, Vertical_Guidance

    State_Vector: TYPE = [# FD           : Flight_Director.State_Vector,
                            LATERAL      : Lateral_Guidance.State_Vector,
                            VERTICAL     : Vertical_Guidance.State_Vector,
                            AP_Engaged? : bool,
                            Overspeed?   : bool                              #]

    %-----------------------------------------------------------------------
    % Map system Flight Director events to internal Flight Director events
    %-----------------------------------------------------------------------
    FD_Event(e: (Flight_Director_Event?)): Flight_Director.Event =
       COND
         AP_Engaged?(e)                -> Turn_On,
         Lateral_Mode_Requested?(e)    -> Turn_On,
         Vertical_Mode_Requested?(e)   -> Turn_On,
         FD_Switch_Hit?(e)             -> Switch,
         Overspeed_Start?(e)           -> Force_Cues
       ENDCOND

    %-----------------------------------------------------------------------
    % Process events that change state external to the FGS
    %-----------------------------------------------------------------------
    Process_External_Event(e: System.Event, s: State_Vector): State_Vector =
       COND
         AP_Engaged?(e)       -> s WITH [AP_Engaged? := true ],
         AP_Disengaged?(e)    -> s WITH [AP_Engaged? := false],
         Overspeed_Start?(e)  -> s WITH [Overspeed?  := true ],
         Overspeed_End?(e)    -> s WITH [Overspeed?  := false],
         ELSE                 -> s
       ENDCOND

    %-----------------------------------------------------------------------
    % Process a flight director event
    %-----------------------------------------------------------------------
    Process_FD_Event(e: System.Event, s: State_Vector):  State_Vector =
     IF Flight_Director_Event?(e) THEN
         LET (newfd, signals) =
              next_state(FD(s), FD_Event(e), AP_Engaged?(s),Overspeed?(s))
         IN IF signals(Turned_Off) THEN
                s WITH [FD        := newfd,
                        LATERAL   := Clear_All_Modes(LATERAL(s)),
                        VERTICAL  := Clear_All_Modes(VERTICAL(s))]
             ELSIF signals(Turned_On) THEN
                 s WITH [FD        := newfd,
                         LATERAL   := Select_Default_Mode(LATERAL(s)),
                         VERTICAL  := Select_Default_Mode(VERTICAL(s)) ]
            ELSE
                s WITH [FD := newfd]
            ENDIF
       ELSE
          s
       ENDIF

    %-----------------------------------------------------------------------
    % Process a lateral or vertical mode event.
    %-----------------------------------------------------------------------
    Process_Flight_Mode_Event(e: System.Event, s: State_Vector): State_Vector =
       IF On?(FD(s)) THEN
          s WITH [LATERAL  := proj_1(next_state(LATERAL(s),e)),
                  VERTICAL := proj_1(next_state(VERTICAL(s),e)) ]
       ELSE
          s
       ENDIF

    %-----------------------------------------------------------------------
    % Next state function
    %-----------------------------------------------------------------------
    next_state(s: State_Vector, e:System.Event): State_Vector =
        Process_Flight_Mode_Event(e,
          Process_FD_Event(e,
            Process_External_Event(e, s)))
```

Figure 31 - PVS Specification of Flight Guidance

Chapter 6

Proofs of Properties

There are many useful properties of the formal model described in Chapter 5 that can be demonstrated with the PVS prover. These include ensuring that important relationships between the modes are maintained, that the system behaves as expected to system events, and even that particular sources of mode confusion do not exist in the model. These proofs are described in this chapter.

6.1 Proving Key Relationships Between the Modes

As discussed in Section 4.1, the architecture of the FGS was chosen to minimize maintenance and support the development of a family of Flight Guidance Systems. This is achieved by breaking the FGS into several small, cohesive components. A consequence of this is that there are several important relationships that must be maintained between the modes. These include:

1. If the autopilot is engaged, the flight director is on.
2. If the flight director is on, one and only one lateral mode is active.
3. If the flight director is on, one and only one vertical mode is active.
4. If the flight director is off, all lateral and vertical modes are cleared.

These properties are stated in PVS in the theory `Flight_Guidance_Properties` shown in Figure 33. All of them are shown by induction over the reachable states. That is, for an arbitrary state `s` and system event `e`, it is shown that if the property holds for state `s`, it also holds for the state `next_state(s,e)`. Since these properties are trivially true of the initial system state, they must hold for all states reachable from the initial state.

For example, property 1 that the flight director must be on if the autopilot is engaged is the first property proven. To simplify stating the desired property, the auxiliary function `FD_On_If_AP_Engaged` is first defined and used in the lemma `FDOIFAPE` (*F*light *D*irector *O*n *IF* Auto*P*ilot *E*ngaged), which states that if `FD_On_If_AP_Engaged` is true of state `s`, it also true of state `next_state(s,e)`. This lemma is proved by the simple PVS proof shown in Figure 32.

```
(" "
(SKOSIMP*)
(LEMMA "System.Event_inclusive")
(INST?)
(APPLY (THEN (SPLIT -1) (GRIND))))
```

Figure 32 - Proof of Lemma FDOIFAPE

```
Flight_Guidance_Properties: THEORY

  BEGIN

    IMPORTING Flight_Guidance,
              Lateral_Guidance_Properties,
              Vertical_Guidance_Properties

    s: VAR Flight_Guidance.State_Vector
    e: VAR System.Event

    %-------------------------------------------------------------------------------
    % The Flight Director is on if the Autopilot is Engaged
    %-------------------------------------------------------------------------------
    FD_On_If_AP_Engaged(s): bool =
       AP_Engaged?(s) => On?(FD(s))

    FDOIFAPE: LEMMA
       FD_On_If_AP_Engaged(s) => FD_On_If_AP_Engaged(next_state(s,e))

    %-------------------------------------------------------------------------------
    % At least one lateral mode is active iff the Flight Director is ON
    %-------------------------------------------------------------------------------
    At_Least_One_Lateral_Mode_Active(s): bool =
       On?(FD(s)) <=> At_Least_One_Mode_Active(LATERAL(s))

    ALOLMA: LEMMA
       At_Least_One_Lateral_Mode_Active(s) =>
          At_Least_One_Lateral_Mode_Active(next_state(s,e))

    %-------------------------------------------------------------------------------
    % There is never more than one lateral mode active.
    %-------------------------------------------------------------------------------
    At_Most_One_Lateral_Mode_Active(s): bool =
       At_Most_One_Mode_Active(LATERAL(s))

    AMOLMA: LEMMA
       At_Least_One_Lateral_Mode_Active(s) &
         At_Most_One_Lateral_Mode_Active(s) =>
           At_Most_One_Lateral_Mode_Active(next_state(s,e))

    %-------------------------------------------------------------------------------
    % At least one vertical mode is active iff the Flight Director is ON
    %-------------------------------------------------------------------------------
    At_Least_One_Vertical_Mode_Active(s): bool =
       On?(FD(s)) <=> At_Least_One_Mode_Active(VERTICAL(s))

    ALOVMA: LEMMA
       At_Least_One_Vertical_Mode_Active(s) =>
          At_Least_One_Vertical_Mode_Active(next_state(s,e))

    %-------------------------------------------------------------------------------
    % At most one vertical mode is active iff the Flight Director is ON
    %-------------------------------------------------------------------------------
    At_Most_One_Vertical_Mode_Active(s): bool =
       At_Most_One_Mode_Active(VERTICAL(s))

    AMOVMA: LEMMA
       At_Least_One_Vertical_Mode_Active(s) &
         At_Most_One_Vertical_Mode_Active(s) =>
           At_Most_One_Vertical_Mode_Active(next_state(s,e))

    %-------------------------------------------------------------------------------
    % A valid state is one in which the Flight Director is on if the Autopilot
    % is engaged and exactly one lateral mode and one vertical mode are
    % active iff the Flight Director is on.
    %-------------------------------------------------------------------------------
    Valid_State(s): bool =
       FD_On_If_AP_Engaged(s)                &
       At_Least_One_Lateral_Mode_Active(s)   &
       At_Most_One_Lateral_Mode_Active(s)    &
       At_Least_One_Vertical_Mode_Active(s)  &
       At_Most_One_Vertical_Mode_Active(s)

    VS: LEMMA
       Valid_State(s) => Valid_State(next_state(s,e))

  END Flight_Guidance_Properties
```

Figure 33 - Flight Guidance Properties

Properties 2, 3, and 4 are shown by proving lemmas ALOLMA (*At Least One Lateral Mode Active*), AMOLMA (*At Most One Lateral Mode Active*), ALOVMA (*At Least One Vertical Mode Active*), and AMOVMA (*At Most One Vertical Mode Active*). These make use of the auxiliary functions in theories Lateral_Guidance_Properties and Vertical_Guidance_ Properties (shown in Figure 34 and Figure 35) that define what it means for at least and at most lateral or vertical mode to be active.

The proof of ALOLMA makes use of the auxiliary predicate At_Least_One_Lateral_ Mode_Active, which is true when either the flight director is off or at least one lateral mode is active. The desired property is then stated as an inductive proof over the reachable states.

The proof of AMOLMA also makes use of an auxiliary predicate At_Most_One_Lateral_ Mode_Active, which is true if no more than one lateral mode is active. The proof of AMOLMA also requires that the predicate At_Least_One_Lateral_Mode_Active holds for state s to eliminate the configurations where the flight director is off and a lateral mode is active. This poses no problems as this predicate was shown to hold for all reachable states in the proof of ALOLMA.

The lemmas ALOVMA and AMOVMA for the vertical modes are similar to those for the lateral modes. Interestingly, the proofs for all four lemmas (ALOLMA, AMOLMA, ALOVMA, and AMOVMA) are identical to those for FDOIFAPE shown in Figure 32.

```
Lateral_Guidance_Properties: THEORY
  BEGIN

    IMPORTING Lateral_Guidance

    s: VAR Lateral_Guidance.State_Vector
    e: VAR (Lateral_Event?)

    %----------------------------------------------------------------------
    % Definition of at least one lateral mode active
    %----------------------------------------------------------------------
    At_Least_One_Mode_Active(s): bool =
       Active?(ROLL(s)) OR
         Active?(HDG(s)) OR
           Active?(NAV(s)) OR
               Active?(GA(s))

   %-----------------------------------------------------------------------
   % Definition of at most one lateral mode active
   %-----------------------------------------------------------------------
   At_Most_One_Mode_Active(s): bool =
       LET R = Active?(ROLL(s)), H = Active?(HDG(s) ,
           N = Active?(NAV(s)),  G = Active?(GA(s)) IN
           (R =>         NOT H & NOT N & NOT G) &
           (H => NOT R         & NOT N & NOT G) &
           (N => NOT R & NOT H         & NOT G) &
           (G => NOT R & NOT H & NOT N        )

  END Lateral_Guidance_Properties
```

Figure 34 - Lateral Guidance Properties

Finally, these properties are combined in the definition of a Valid_State, i.e., a state in which all of these properties hold. Lemma VS asserts that if Valid_State holds in state s, it also holds in state next_state(s,e). The proof of VS makes direct use of the proofs of lemmas FDOIFAPE, ALOLMA, AMOLMA, ALOVMA, and AMOVMA.

6.2 Regression Analysis

The modular architecture of the FGS facilitates incremental development of the PVS specification by adding new modes to the existing framework. This was precisely the pattern we followed in developing both the ObjecTime and PVS models. While it was straightforward to specify the behavior of a single mode such as ROLL or GA, we found ourselves making small mistakes when modifying the Lateral_Guidance, Vertical_Guidance, or Flight_Guidance theories to include these new modes. As a result, we started developing simple putative lemmas that described the response of the FGS to each new system event. A few of these are shown in Figure 36.

Most of these could be proved with a single PVS (GRIND) command. While not technically deep or challenging, they were very useful for checking that an error had not been introduced in an already completed portion of the model. This process was very similar to regression testing, except that instead of running test cases after each change, we ran the proofs of these lemmas.

```
Vertical_Guidance_Properties : THEORY

  BEGIN

    IMPORTING Vertical_Guidance

    s: VAR Vertical_Guidance.State_Vector
    e: VAR (Vertical_Event?)

    %-----------------------------------------------------------------------
    % Definition of at least one vertical mode active
    %-----------------------------------------------------------------------
    At_Least_One_Mode_Active(s): bool =
       Active?(PITCH(s)) OR
         Active?(VS(s)) OR
           Active?(GA(s))

    %-----------------------------------------------------------------------
    % Definition of at most one vertical mode active
    %-----------------------------------------------------------------------
    At_Most_One_Mode_Active(s): bool =
       LET P = Active?(PITCH(s)), V = Active?(VS(s)), G = Active?(GA(s))  IN
            (P =>        NOT V & NOT G) &
            (V => NOT P        & NOT G) &
            (G => NOT P & NOT V       )

  END Vertical_Guidance_Properties
```

Figure 35 - Vertical Guidance Properties

```
Flight_Guidance_Checks: THEORY

  BEGIN

    IMPORTING Flight_Guidance_Properties

    s: VAR Flight_Guidance.State_Vector
    e: VAR System.Event

    %------------------------------------------------------------------------
    % Check for correct response to pressing HDG button.
    %------------------------------------------------------------------------
    HDG_Selected: LEMMA
      Valid_State(s) &
        NOT ACTIVE?(state(HDG(LATERAL(s)))) & HDG_Switch_Hit?(e) =>
            ACTIVE?(state(HDG(LATERAL(next_state(s,e)))))

    HDG_Deselected: LEMMA
      Valid_State(s) &
        ACTIVE?(state(HDG(LATERAL(s)))) & HDG_Switch_Hit?(e) =>
          ACTIVE?(state(ROLL(LATERAL(next_state(s,e)))))

    %------------------------------------------------------------------------
    % Check for correct response to pressing NAV button.
    %------------------------------------------------------------------------

    NAV_Selected: LEMMA
      Valid_State(s) &
        NOT ACTIVE?(state(NAV(LATERAL(s)))) & NAV_Switch_Hit?(e) =>
            ACTIVE?(state(NAV(LATERAL(next_state(s,e)))))

    NAV_Deselected: LEMMA
      Valid_State(s) &
        ACTIVE?(state(NAV(LATERAL(s)))) & NAV_Switch_Hit?(e) =>
          ACTIVE?(state(ROLL(LATERAL(next_state(s,e)))))

                                        . . .           . . .

    %------------------------------------------------------------------------
    % Check for correct response to pressing VS button.
    %------------------------------------------------------------------------
    VS_Selected: LEMMA
      Valid_State(s) &
        NOT ACTIVE?(state(VS(VERTICAL(s)))) & VS_Switch_Hit?(e) =>
            ACTIVE?(state(VS(VERTICAL(next_state(s,e)))))

    VS_Deselected: LEMMA
      Valid_State(s) &
        ACTIVE?(state(VS(VERTICAL(s)))) & VS_Switch_Hit?(e) =>
          ACTIVE?(state(PITCH(VERTICAL(next_state(s,e)))))

    %------------------------------------------------------------------------
    % Check for correct response to pressing the FD button.
    %------------------------------------------------------------------------
    FD_OFF: LEMMA
       OFF?(state(FD(s))) & FD_Switch_Hit?(e) =>
          CUES?(state(FD(next_state(s,e))))

    FD_ON: LEMMA
       ON?(state(FD(s))) & FD_Switch_Hit?(e) &
          NOT (AP_Engaged?(s) OR Overspeed?(s)) =>
              OFF?(state(FD(next_state(s,e))))

    FD_CUES: LEMMA
       CUES?(state(FD(s))) & FD_Switch_Hit?(e) &
          (AP_Engaged?(s) OR Overspeed?(s)) =>
             NO_CUES?(state(FD(next_state(s,e))))

   FD_NO_CUES: LEMMA
       NO_CUES?(state(FD(s))) & FD_Switch_Hit?(e) &
          (AP_Engaged?(s) OR Overspeed?(s)) =>
              CUES?(state(FD(next_state(s,e))))

  END Flight_Guidance_Checks
```

Figure 36 - Checks of FGS Response to System Events

6.3 Searching for Sources of Mode Confusion

As discussed in Section 2.1, Leveson, et.al. [13], identify six categories of design that have historically been sources of mode confusion:

1. Interface interpretation errors
2. Inconsistent behavior
3. Indirect mode changes
4. Operator authority limits
5. Unintended side effects
6. Lack of appropriate feedback

To the extent that these can be expressed formally, automated tools can be used to systematically determine if such sources of mode confusion exist in our models. While much work remains to be done in this area, this section demonstrates this concept with a few examples. The purpose of this section is only to show how automated analysis can be used to discover potential sources of mode confusion. Once such examples are discovered, there still needs to be a careful review of their potential for mode confusion.

6.3.1 Inconsistent Behavior

Precisely defining the concept of inconsistent behavior is nontrivial and likely to be a long term endeavor. However, examples of inconsistent behavior can easily be specified formally and verified. For example, in Section 4.2 it was shown that many of the switches in the FGS act as toggles, switching a mode between its CLEARED and ACTIVE states. But do the switches act as toggles in all possible states? This question can be answered by proving a few simple lemmas about each switch. For example, to assert that the HDG switch behaves as a toggle, we create the lemmas

```
HDG_Toggle_1: LEMMA
       NOT Active?(HDG(LATERAL(s))) =>
           Active?(HDG(LATERAL(next_state(s, HDG_Switch_Hit))))

HDG_Toggle_2: LEMMA
           Active?(HDG(LATERAL(s))) =>
       NOT Active?(HDG(LATERAL(next_state(s, HDG_Switch_Hit))))
```

Each of these are easily proved with a single PVS Grind command. Similar lemmas are shown for the other switches in theory `Consistent_Behavior_Checks` (Figure 37). While the HDG, NAV, and VS switches always behave as toggles in the current model, this will not hold as the model is expanded to include more modes. For example, in the CoRE FGS specification [15] the VS switch is inhibited when in the Track state of Vertical Approach mode. When Vertical Approach is added to the PVS model, the lemmas about the affected switches will need to be changed to describe their new behavior. However, these lemmas serve as quick and easy check on the model as new modes are incorporated.

```
Consistent_Behavior_Checks: THEORY
  BEGIN

    IMPORTING Flight_Guidance_Properties

    s: VAR Flight_Guidance.State_Vector
    e: VAR System.Event

    %-----------------------------------------------------------------------
    % Lemmas used to check that switches always act as toggles
    %-----------------------------------------------------------------------
    HDG_Toggle_1: LEMMA
       NOT Active?(HDG(LATERAL(s))) =>
           Active?(HDG(LATERAL(next_state(s, HDG_Switch_Hit))))

    HDG_Toggle_2: LEMMA
           Active?(HDG(LATERAL(s))) =>
       NOT Active?(HDG(LATERAL(next_state(s, HDG_Switch_Hit))))

    NAV_Toggle_1: LEMMA
       NOT Active?(NAV(LATERAL(s))) =>
           Active?(NAV(LATERAL(next_state(s, NAV_Switch_Hit))))

    NAV_Toggle_2: LEMMA
           Active?(NAV(LATERAL(s))) =>
       NOT Active?(NAV(LATERAL(next_state(s, NAV_Switch_Hit))))

    VS_Toggle_1: LEMMA
       NOT Active?(VS(VERTICAL(s))) =>
           Active?(VS(VERTICAL(next_state(s, VS_Switch_Hit))))

    VS_Toggle_2: LEMMA
           Active?(VS(VERTICAL(s))) =>
       NOT Active?(VS(VERTICAL(next_state(s, VS_Switch_Hit))))

END Consistent_Behavior_Checks
```

Figure 37 - Consistent Behavior Checks

Also note that the GA switch is not included in this list. The GA switch, which is mounted on the Control Yoke rather than the Flight Control Panel, behaves differently from the other switches in that it only selects Go Around mode. To deselect Go Around mode, the pilot must select another mode, press the SYNC button, or engage the Autopilot.

6.3.2 Ignored Crew Inputs

Direct inputs from the flight crew that are ignored by the automation in some states are likely to be potential sources of mode confusion. Unfortunately, it can be very difficult to determine all the cases in which a crew input is ignored. However, if these concepts can be formalized, the conditions under which a crew input is ignored can easily be found using our model.

To do this, we first define the concept of a crew input. A list of events that meet our informal notion of a crew input are easily identified by scanning the list of system events. These are enumerated by defining a predicate `Crew_Input?` over the system events. This predicate is shown in theory `Ignored_Crew_Inputs` in Figure 38. Next, we define what is means for a crew input to be ignored. For this example, we define this as the failure of an event to cause a mode change. The predicate `Mode_Change?`, shown in theory `Ignored_Crew_Inputs` in Figure 38, defines a mode change as change in state of the Flight Director or one of the lateral or vertical modes.

```
Ignored_Crew_Inputs: THEORY
  BEGIN

    IMPORTING Flight_Guidance_Properties

    s: VAR Flight_Guidance.State_Vector
    e: VAR System.Event

    %------------------------------------------------------------------------
    % Events directly initiated by the flight crew
    %------------------------------------------------------------------------
    Crew_Input?(e: System.Event) : bool =
         AP_Engaged?(e)                 OR
         SYNC_Switch_Pressed?(e)        OR
         SYNC_Switch_Released?(e)       OR
         FD_Switch_Hit?(e)              OR
         Lateral_Mode_Requested?(e)     OR
         Vertical_Mode_Requested?(e)    OR
         VS_Pitch_Wheel_Changed?(e)

    %------------------------------------------------------------------------
    % A mode change occurs when the state of the autopilot, flight director,
    % or any lateral or vertical mode changes.
    %------------------------------------------------------------------------
    Mode_Change?(s,e): bool =
      state(FD(s))              /= state(FD(next_state(s,e)))                OR
      state(ROLL(LATERAL(s)))   /= state(ROLL(LATERAL(next_state(s,e))))    OR
      state(HDG(LATERAL(s)))    /= state(HDG(LATERAL(next_state(s,e))))     OR
      state(NAV(LATERAL(s)))    /= state(NAV(LATERAL(next_state(s,e))))     OR
      state(GA(LATERAL(s)))     /= state(GA(LATERAL(next_state(s,e))))      OR
      state(PITCH(VERTICAL(s))) /= state(PITCH(VERTICAL(next_state(s,e)))) OR
      state(VS(VERTICAL(s)))    /= state(VS(VERTICAL(next_state(s,e))))     OR
      state(GA(VERTICAL(s)))    /= state(GA(VERTICAL(next_state(s,e))))

    %------------------------------------------------------------------------
    % Lemma used to search for ignored crew inputs
    %------------------------------------------------------------------------
    Search_For_Ignored_Crew_Inputs: LEMMA
       Valid_State(s) & Crew_Input?(e) => Mode_Change?(s,e)

    %------------------------------------------------------------------------
    % Crew inputs that do not cause a mode change.
    %     o Engaging the Autopilot when not in Go Around mode
    %     o Pressing the GA Switch when in Go Around mode
    %     o Pressing the SYNC switch when not in Go Around mode
    %     o Releasing the SYNC switch
    %     o Rotating the Vertical Speed/Pitch Wheel when Flight Director is off
    %     o Rotating the Vertical Speed/Pitch Wheel when in Pitch mode
    %------------------------------------------------------------------------
    Ignored_Crew_Input?(s,e): bool =
         AP_Engaged?(e)              &
           NOT (Active?(GA(LATERAL(s))) or Active?(GA(VERTICAL(s))))    OR
         GA_Switch_Hit?(e)       &
                (Active?(GA(LATERAL(s))) &  Active?(GA(VERTICAL(s))))   OR
         SYNC_Switch_Pressed?(e)     &
           NOT (Active?(GA(LATERAL(s))) or Active?(GA(VERTICAL(s))))    OR
         SYNC_Switch_Pressed?(e)     & NOT (On?(FD(s)))                 OR
         SYNC_Switch_Released?(e)                                       OR
         VS_Pitch_Wheel_Changed?(e)  & NOT (On?(FD(s)))                 OR
         VS_Pitch_Wheel_Changed?(e)  &
            (Active?(PITCH(VERTICAL(s))))

    %------------------------------------------------------------------------
    % Lemma used to confirm that all ignored crew inputs are known
    %------------------------------------------------------------------------
    No_Known_Ignored_Crew_Inputs: LEMMA
       Valid_State(s) &
         Crew_Input?(e) &
           NOT Ignored_Crew_Input?(s,e) => Mode_Change?(s,e)

END Ignored_Crew_Inputs
```

Figure 38 - Ignored Crew Inputs

To search for crew inputs that are ignored, we assert the (false) lemma

```
Search_For_Ignored_Crew_Inputs: LEMMA
      Valid_State(s) & Crew_Input?(e) => Mode_Change?(s,e)
```

and try to prove it with the PVS prover. This results in several proof sequents (e.g., proof obligations) such as

```
{-1}    VS_Pitch_Wheel_Changed?(e!1)
[-2]    Valid_State(s!1)
  |-------
{1}    CUES?(state(FD(s!1)))
{2}    NO_CUES?(state(FD(s!1)))
```

The antecedents {-1} and {-2} are the assertions known to be true at this point in the proof. To complete this sequent, we have to show that either of the consequents {1} or {2} follow from the antecedents. That is, given that the event e!1 is that the VS/Pitch Wheel was rotated, and that we start in a valid state s!1, we must prove that the Flight Director was in the state CUES or NO CUES prior to rotating the VS/Pitch Wheel. Since our model provides no such constraint between the state of the Flight Director and our ability to rotate the VS/Pitch Wheel, this cannot be proven.

At this point, we realize that the reason PVS is requiring us to prove the impossible is because our original lemma was false. If the FD is in the state OFF, rotation of the VS/Pitch Wheel does not cause a mode change in the FGS. In other words, we have found a case where a crew input does not cause a mode change. Examination of the other sequents generated by PVS leads the following list of seven crew inputs that are ignored by the FGS: [5]

1. The Autopilot is engaged while not in lateral or vertical Go Around mode
2. The GA switch is pressed while in lateral and vertical Go Around mode
3. The SYNC switch is pressed while not in lateral or vertical Go Around mode
4. The SYNC switch is pressed while the Flight Director is turned off
5. The SYNC switch is released
6. The VS/Pitch Wheel is rotated while the Flight Director is turned off
7. The VS/Pitch Wheel is rotated while in Pitch mode

[5] The astute reader will note that rotation of the VS/Pitch Wheel while in VS mode is not included in the list. At the end of Phase I, this scenario causes a mode change to PITCH in the PVS model. The necessary constraint will be added in Phase II of the project.

To avoid having to reexamine this list of sequents each time the model is changed, we define a predicate `Ignored_Crew_Inputs?` (shown in Figure 38) that enumerates each of these cases. We then create the lemma

```
No_Known_Ignored_Crew_Inputs: LEMMA
    Valid_State(s) &
       Crew_Input?(e) &
          NOT Ignored_Crew_Input?(s,e) => Mode_Change?(s,e)
```

that takes this list of known ignored crew inputs into account. As expected, the proof of this lemma completes (in a little over a minute).

Trying to prove a lemma suspected to be false has allowed us to generate an explicit list of ignored crew inputs. Used in this way, PVS becomes a tool of discovery rather than verification. Moreover, the final proof can be rerun each time the PVS model of the mode logic is changed to ensure no additional ignored crew inputs are created.

6.3.3 Indirect Mode Changes

Another common source of mode confusion identified in [13] is indirect mode changes. Indirect mode changes occur when the system changes mode without a direct input from the operator. While indirect mode changes are usually evident by inspection, the same technique used in the previous section to detect ignored user inputs can be used to detect indirect mode changes. In this case the lemma that we need to try to prove is

```
Search_For_Indirect_Mode_Changes: LEMMA
       Valid_State(s) & NOT Crew_Input?(e) => NOT Mode_Change?(s,e)
```

This lemma states that all events that are not crew inputs, i.e., that are not direct inputs from the operator, do not cause a mode change. Clearly, the exceptions to this lemma will be indirect mode changes. Attempting to prove this lemma yields several unprovable sequents, just as in Section 6.3.2. Examination of these leads to the following three sources of indirect mode changes in the PVS model:

1. The overspeed condition becomes true while the Flight Director is turned off or is not displaying the guidance cues

2. NAV mode remains Armed for the required minimum time

3. Navigation source is captured while in the Armed state of NAV mode

The first of indirect mode change forces the Flight Director to be turned on or display the guidance cues when the overspeed condition occurs. The second causes a transition from the ARMED INITIAL state of NAV mode to the ARMED LONG ENOUGH state of NAV mode. [6]

[6] This transition would be almost impossible for the flight crew to detect and probably should not even be considered a mode change.

```
Indirect_Mode_Changes: THEORY
BEGIN

    IMPORTING Ignored_Crew_Inputs

    s: VAR Flight_Guidance.State_Vector
    e: VAR System.Event

    %--------------------------------------------------------------------------
    % Lemma used to search for indirect mode changes
    %--------------------------------------------------------------------------
    Search_For_Indirect_Mode_Changes: LEMMA
       Valid_State(s) & NOT Crew_Input?(e) => NOT Mode_Change?(s,e)

    %--------------------------------------------------------------------------
    % The only mode changes not caused by crew inputs are
    %    o Overspeed while Flight Director off or not displaying guidance cues
    %    o NAV armed minimum reached while in NAV Armed Initial state
    %    o NAV track condition met while in NAV Armed Long Enough state
    %--------------------------------------------------------------------------
    Indirect_Mode_Change?(s,e): bool =
         Overspeed_Start?(e)         &
           NOT (CUES?(state(FD(s))))              OR
         NAV_Armed_Long_Enough?(e)     &
           ARMED_INITIAL?(state(NAV(LATERAL(s))))  OR
         NAV_Track_Cond_Met_Event?(e) &
            ARMED_LONG_ENOUGH?(state(NAV(LATERAL(s))))

    %--------------------------------------------------------------------------
    % Lemma used to ensure all indirect mode changes are known
    %--------------------------------------------------------------------------
    No_Unknown_Indirect_Mode_Changes: LEMMA
       Valid_State(s) &
           NOT Crew_Input?(e) &
               NOT Indirect_Mode_Change?(s,e) => NOT Mode_Change?(s,e)

END Indirect_Mode_Changes
```

Figure 39 - Ignored Crew Inputs

The last would cause a change from the ARMED to TRACK state of NAV mode when the navigation source is captured.

As was done for ignored crew inputs, we define a predicate that enumerates the indirect mode changes and create a new lemma to be proven:

```
No_Unknown_Indirect_Mode_Changes: LEMMA
        Valid_State(s) &
             NOT Crew_Input?(e) &
                  NOT Indirect_Mode_Change?(s,e) => NOT Mode_Change?(s,e)
```

More indirect mode changes will be introduced as the PVS model is expanded. This lemma will make it simple to maintain an explicit list of all indirect mode changes.

Chapter 7
Conclusions and Future Directions

7.1 Conclusions

This project has explored ways to detect mode confusion through deeper scrutiny of the behavior of the automation. This approach makes use of two complementary strategies. The first is to create a clear, executable model of the automation, connect it to a simulation of the flight deck, and use this combination to review of the behavior of the automation and the man-machine interface with engineers, the pilots, and experts in human factors. The second is to conduct mathematical analyses of the model. In addition, the models and visualizations are consistent with an architecture that supports a product family approach to the development of Flight Guidance Systems.

7.1.1 Visualization of the Automation

Creating a clear model of the automation that can be connected to a simulation of the flight deck and executed has several benefits. One of the most important is to provide a common focus that facilitates discussion between pilots, experts in human factors, and the system designers. Chapter 4 attempts to illustrate the value of this with a few examples. Unfortunately, static examples cannot really convey the utility of stepping through the simulation with an audience of experts from different disciplines. Our experiences to date have shown that, if anything, we underestimated the power of this technique. In every demonstration, the visualization has generated vigorous, positive debate between these groups.

A secondary benefit is to force the development and commitment to a high level design of the automation. It is our belief that this leads naturally to simpler systems. In projects developed without such a vision, design choices may be based on local concerns, such as fixing the immediate problem at hand or achieving a certain level of performance, and this tends to result in unnecessary complexity that is confusing to both the users and developers of the system. Moreover, this complexity tends to grow as the system evolves over time. Having a clear, high level model of the automation encourages the developers (and customers) to make changes consistent with this model as time progresses.

We do not yet know if a dynamic, high level model of the automation would be of value during training of the flight crew. The few times we have demonstrated the simulation to pilots, they have tended to focus on the details of the particular flight control system and have been noncommittal to its potential for use during training. We suspect that they would be much more interested in the simulation if it described an actual airplane they fly rather than the example described in [15].

One benefit we had not anticipated at the start of the project was the value of creating a model of the automation specifically tailored for conveying an accurate mental model of the automation. Early in the project, it became clear that a full executable specification of the automation was not appropriate for conveying a mental model of the automation. Instead, we focused on creating a visualization of the automation (Figure 5) that conveyed what we felt was the most relevant information. While this model was a true abstraction of the automation (i.e., every property of the model was also a property of the automation), it left out many details that would be needed to actually implement the mode logic in order to not obscure the key ideas. As our experience with the model grew, we began to realize that there might be additional information that could be incorporated into the visualization to help convey the appropriate mental model. For example, one reviewer suggested coloring the transitions last executed a different color from the others.

7.1.2 Automated Analysis of the Model

At the start of the project, it was not clear how useful automated analyses, especially using the PVS theorem prover, would be in detecting sources of mode confusion. However, as the project progressed, we found ourselves relying more upon the PVS models and our ability to prove properties and less upon the executable ObjecTime models and the visualizations.

As described in Section 4.1, our goal of adopting an architecture for a family of Flight Guidance Systems lead naturally to a specification consisting of many, small, reusable components. The ability to prove key relationships between these components, such as only one lateral mode is active at a time (Section 6.1), was very helpful. Component based development (and specification) appears to be on the rise as companies try to find ways to reduce costs through systematic reuse and product family development. While this makes specification of the smaller components simpler, it increases the need to verify properties of the overall system. This suggests that component based approaches, including those found in many object-oriented methods, increase rather than decrease the need for formal analysis.

We found that combining the proofs of the key relationships along with the proofs of many simple properties provided us with a valuable regression suite. After making any change to the model, such as adding a new mode, we routinely ran the entire suite of proofs. More often than not, one or more proofs would no longer go through to completion. This then led us to inspect the model to determine if that property should still be true and if so, why it no longer held. This ability to automatically check for behavior that we would normally check manually was an easy way to maintain a high level of confidence in our model.

Using PVS to detect sources of mode confusion was one of the most novel aspects of the project. Using PVS to find all states in which crew inputs are ignored (Section 6.3.2) and all indirect mode changes (Section 6.3.3) illustrates to how a wide variety of sources of mode confusion can be detected in such models. The central question here is which potential sources of mode confusion can be described formally. For example, it is not clear how the notion of inconsistent behavior can be stated formally, but in Section 6.3.1 we discussed how a few simple forms of inconsistent behavior can be formalized. We believe there is room for much more work in this area.

7.1.3 Support for Product Families

It is notable that the ObjecTime models, the visualizations of the mode logic, and the PVS models are all consistent with a product family architecture designed to accommodate the most important variations found in Flight Guidance Systems, the configuration of modes installed on the aircraft. The architecture described in Section 4.1 and that described in [15] are considerably different, even though their overall behavior is similar. It is our belief that the architecture described here not only supports product family variations better, but is a clearer and more intuitive representation of the mode logic.

We find it intriguing that the same architecture can support both a clear mental model of the automation and product family variations. At least on the surface, there is no obvious connection between these two goals. However, if most variations result from the different ways in which a product is used in its environment, it may be that the simplest mental model is naturally compatible with an architecture that supports these variations. In this one example, that seems to be the case.

In any case, if complex systems are to be affordable, planning for change and reuse has to play a larger role in the future, and the simplest way to achieve this is by developing an architecture for an entire family of products. Since a mental model of the automation must be an abstraction of the actual implementation, i.e., every property of the model must also be true of the implementation, it is encouraging that the goals of providing a clear mental model and a product family architecture do not appear to be in conflict.

In contrast, the architecture described in Section 4.1 did make the proofs more difficult to complete. However, this could always be overcome by first proving lemmas about the overall system state that could then be used in the main proofs.

7.2 Future Directions

There are several areas for future work. These include extending the existing models with more modes, developing more analyses for mode confusion, integrating the properties of flight control laws into the models, investigating alternative architectures for the FGS, extensions to the visualization to convey an accurate mental model of the automation, and making extensions to PVS to simplify modeling.

7.2.1 Extending the Models

Only part of the mode logic described in the CoRE FGS specification [15] was modeled in Phase I. In particular, the complex interactions between Vertical Approach and the other modes, especially lateral Approach, were not completed. Altitude Select mode has not yet been modeled, precisely because it is one of the more complex vertical modes that imposes several constraints between it and the other vertical modes. Even in the modes currently modeled, not all the constraints between them have been specified. For example, the constraint that the system can be in lateral Go Around mode if and only if it is in vertical Go Around mode is missing. All of these are areas for further work.

Beyond this, the original CoRE FGS specification did not include the complex vertical navigation modes found in modern Flight Guidance Systems. These are known to be one of the main sources of mode confusion [9]. Extending the model with modes beyond those in CoRE specification is another area for further work.

7.2.2 Extending the Analyses

In [13], the authors identify six common sources of mode confusion. In Section 6.3 we attempted to formally state of few of these and use the PVS prover to detect instances of them in our models. However, much more work remains to be done in this area. It should be possible to formally characterize many more potential sources of mode confusion and search for them using automated tools.

7.2.3 Integration of Flight Control Laws

The models described here do not include the flight control laws that actually generate the flight guidance commands. Incorporating this information into the models may be necessary to investigate some forms of mode confusion. The crash of an Airbus A330-322 in Toulouse, France on 6/30/1994 illustrates this claim [2]. During a flight test of a simulated engine failure, an unexpected mode transition to altitude acquisition (ALT*) occurred. Pitch protection was not provided in ALT* mode, although it was present in all of the other modes. Detection of inconsistent behavior such as this will require elaboration of the basic properties of the control laws in addition to the mode structure.

7.2.4 Investigation of Alternative Architectures

The architecture for the models presented in this report have assumed that the coupling between the lateral and vertical guidance modes is minimal. However, some lateral and vertical modes are more closely coupled than others. For example, Section 4.2 describes how the lateral and vertical Approach modes are closely synchronized. In some aircraft, the lateral and vertical Go Around are effectively a single mode since if either is active, the other must also be active.

As discussed in Section 2.1, IA. A. Lambregts, FAA National Resource Specialist for Advanced Controls, argues that much of the complexity of the flight deck derives from the independent design of the autopilot and the autothrottle [11]. Addressing these concerns could require both integration of the properties of the flight control laws and the investigation of alternative architectures.

7.2.5 Visualization of the Automation

In Section 7.1.1 we discussed the unanticipated benefit of creating a visualization of the automation specifically tailored for conveying an accurate mental model of the automation. Not only did we decide that there was value in leaving some information out of the visualization in order to not obscure the main points, but that there was value in including some information that would not normally be included in a requirements or design model. Precisely what information should be omitted or included in the visualization to best convey a useful mental model of the automation is yet another topic for future work.

7.2.6 Extensions to PVS

We occasionally found the goals of creating a product family architecture and specifying the mode logic in PVS to be in conflict. While specification of the simpler components, such as `Simple_Guidance` and `Arming_Guidance`, was straightforward, the "glue" theories such as `Lateral_Guidance`, `Vertical_Guidance`, and `Flight_Guidance` were much more difficult. We attribute much of this to the lack of domain specific constructs for communication between components. For example, notations such as ObjecTime [21] and SCR [7] provide explicit constructs for communication between components and determining the order in which events are processed. Since PVS is a general purpose notation, comparable capabilities had to be explicitly constructed in the glue theories. For example, our use of events and signals is a rudimentary communications mechanism between the components of the FGS. Developing a reusable infrastructure in PVS to facilitate assembling component specifications into an overall system specification, much as was done in [19] for SCR, is another area we would like to investigate.

To achieve a product family architecture, the FGS model makes extensive use of information hiding. For example, the states of a mode, which states are active, and the transitions between states are encapsulated in the `Simplified_Guidance` and `Arming_Guidance` theories. At each level, the synchronization between components is specified in to the parent component to ensure that siblings had no information about their peers. However, all of this was done implicitly. In Phase II, we would like to investigate if the information hiding capabilities of PVS can be used to enforce this explicitly.

The lack of state variables in PVS also made creating the models more difficult. In the FGS, system state was modeled as a record structure that mirrors the FGS architecture. Evaluation of a system function also mirrors the FGS architecture, with a function in a component recursively calling functions in its child components. At each step, the appropriate component of state is extracted and passed as a parameter to the child function. If the function returns a state, the state is reconstructed as the evaluation returns. While state variables would probably make proofs more complicated, it would be much more natural and intuitive to embed state variables in the appropriate components.

Bibliography

[1] Kathy Abbott, Presentation Slides, FAA Autoflight Mode Awareness Workshop, Seattle, WA, June 14-16, 1998.

[2] Charles E. Billings. Aviation Automation: the Search for a Human Centered Approach, Lawrence Erlbaum Associates, Inc., Mahwah, NJ, 1997.

[3] Mark Blackburn and Joseph Fontaine, Specification Transformation to Support Automated Testing, Technical Report SPC-97036-MC, Software Productivity Consortium, April 1998.

[4] Lisa Brownsword and Paul Clements, A Case Study in Successful Product Line Development, Report CMU/SEI-96-TR-016, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213, October 1996.

[5] Francis Fung and Damir Jamsek, Formal Specification of a Flight Guidance System, NASA/CR-1998-206915, January 1998.

[6] P. Heimdahl and Nancy G. Leveson, Completeness and Consistency in Hierarchical State-Based Requirements, IEEE Transactions on Software Engineering, 22(6):363-377, June 1996.

[7] Constance L. Heitmeyer, James Kirby, and Bruce G. Labaw, Automated Consistency Checking of Requirements Specification, ACM Transactions on Software Engineering and Methodology (TOSEM), 5(3):231-261, July 1996.

[8] Dan Hughes, Glass Cockpit Study Reveals Human Factor Problems, Aviation Week & Space Technology, August 7, 1989.

[9] Dan Hughes and Michael Dornheim, Automated Cockpits: Who's in Charge?, Aviation Week & Space Technology, January 30-February 6, 1995

[10] George Krasovec, Natarajan Shankar, and Paul Ward, Integration of Formal Verification with Real-Time Design, Advanced System Technologies, Englewood, CO.

[11] A. A. Lambregts, Automatic Flight Controls: Concepts and Methods, Draft Report, FAA National Resource Specialist for Advanced Controls, 1998.

[12] Nancy G. Leveson, Safeware: System Safety and Computers, Addison-Wesley Publishing Company: Reading, Massachusetts, 1995.

[13] Nancy Leveson, et al, Analyzing Software Specifications for Mode Confusion Potential, Presented at the Workshop on Human Error and System Development, Glasgow, http://www.cs.washington.edu/research/hci/, March 1997.

[14] Steven P. Miller and Mandayam Srivas, Formal Verification of the AAMP5 Microprocessor, Workshop on Industrial-Strength Formal Specification Techniques (WIFT95), April 5-8, Boca Raton, Florida, 1995.

[15] Steven P. Miller and Karl F. Hoech, Specifying the mode logic of a flight guidance system in CoRE, Rockwell Technical Report WP97-2011, Rockwell Collins, November 1997.

[16] Steven P. Miller, Specifying the Mode Logic of a Flight Guidance System in CoRE and SCR, Second Workshop on Formal Methods in Software Practice (FMSP98), March 4-5, Clearwater Beach, Florida, 1998.

[17] Dimitri Naydich and John Nowakowski, Flight Guidance System Validation Using SPIN, NASA/CR-1998-208432, June 1998.

[18] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal Verification for Fault-tolerant Architectures: Prolegomena to the Design of PVS. IEEE Transactions on Software Engineering, 21(2):107-125, Feb. 1995.

[19] Sam Owre, John Rushby, and Natarajan Shankar, Analyzing Tabular and State-Transition Requirements Specifications in PVS, NASA CR-201729, July 1997.

[20] N.B. Sarter and D.D. Woods. Decomposing Automation: Autonomy, Authority, Observability and Perceived Animacy. First Automation Technology and Human Performance Conference, April 1994.

[21] Bran Selic, G. Gullekson, and P. Ward, Real-Time Object-Oriented Modeling, John Wiley & Sons, 1994.

[22] Software Productivity Consortium, Reuse-Driven Software Processes Guidebook, Report SPC-92019-CMC, SPC Building, 2214 Rock Hill Road, Herndon, VA 22070, November 1993.

[23] Earl F. Weener, Commercial Transport Safety, Airliner, Boeing Commercial Airplane Group, pg. 1-9, April-June 1993.

[24] David M. Weiss, Defining Families: The Commonality Analysis, Lucent Technologies Bell Laboratories, 1000 E. Warrenville Rd, Napierville, IL, 60566, 1997.

REPORT DOCUMENTATION	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE January 1999	3. REPORT TYPE AND DATES COVERED Contractor Report

4. TITLE AND SUBTITLE Detecting Mode Confusion Through Formal Modeling and Analysis	5. FUNDING NUMBERS NAS1-19704 522-33-31-01
6. AUTHOR(S) Steven P. Miller and James N. Potts	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Rockwell Colliins, Inc. Advanced Technology Center Cedar Rapids, IA 52402	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Langley Research Center Hampton, VA 23681-2199	10. SPONSORING / MONITORING AGENCY REPORT NUMBER NASA/CR-1999-208971
11. SUPPLEMENTARY NOTES Langley Technical Monitor: Ricky Butler Phase I Final Report	
12a. DISTRIBUTION / AVAILABILITY STATEMENT Unclassified-Unlimited Subject Category: 61 Distribution: Standard Availability: NASA CASI (301) 621-0390	12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

Aircraft safety has improved steadily over the last few decades. While much of this improvement can be attributed to the introduction of advanced automation in the cockpit, the growing complexity of these systems also increases the potential for the pilots to become confused about what the automation is doing. This phenomenon, often referred to as mode confusion, has been involved in several accidents involving modern aircraft. This report describes an effort by Rockwell Collins and NASA Langley to identify potential sources of mode confusion through two complementary strategies. The first is to create a clear, executable model of the automation, connect it to a simulation of the flight deck, and use this combination to review of the behavior of the automation and the man-machine interface with the designers, pilots, and experts in human factors. The second strategy is to conduct mathematical analyses of the model by translating it into a formal specification suitable for analysis with automated tools. The approach is illustrated by applying it to a hypothetical, but still realistic, example of the mode logic of a Flight Guidance System.

14. SUBJECT TERMS formal methods, flight guidance systems, mode confusion, software verification	15. NUMBER OF PAGES 69
	16. PRICE CODE A04

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102